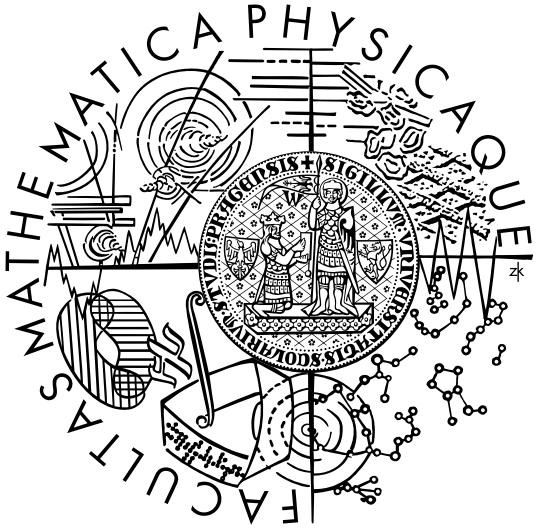
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jakub Galgonek

Dotazovací jazyky pro Sémantický web

Query languages for the Semantic web

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
Studijní program: Informatika, softwarové systémy

2008

Děkuji RNDr. Jakubu Yaghobovi, Ph.D. za vedení práce a Mgr. Jiřímu Dokulilovi za pomoc s proniknutím do zdrojového kódu datového úložiště Trisolda.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 8.8.2008 Jakub Galgonek

Obsah

1 Úvod **1**

2 Dotazovací jazyky pro Sémantický web **2**
2.1 Jazyk metadat . . . 2
2.2 Výběr dotazovacích jazyků . . . 3
2.3 Primární účel dotazovacího jazyka . . . 3
2.4 Podpora slovníků . . . 5
2.5 Původ jazyků . . . 6
2.6 Výběr dat z RDF databáze . . . 8
2.7 Nepovinná data . . . 14
2.8 Přístup k anonymnímu uzlu . . . 15
2.9 Vytváření RDF dat . . . 16
2.10 Výběr dat s rekurzivně definovanou strukturou . . . 18
2.11 Podpora odvozování . . . 20
2.12 Podpora datatypů . . . 22
2.13 Možnosti jazyků obecně . . . 22

3 Návrh jazyka **24**
3.1 Volba přístupu . . . 24
3.2 Základ jazyka . . . 25
3.3 Pojmenované vzory a výběr dat . . . 26
3.4 Vyhodnocování dotazů . . . 28
3.5 Vytváření nových dat . . . 33
3.6 Vytváření strukturovaných dat . . . 34
3.7 Skládání dotazů . . . 36
3.8 Výrazy . . . 37
3.9 Práce s anonymními uzly . . . 38
3.10 Moduly . . . 40
3.11 Syntaktický cukr . . . 40
3.11.1 Vzor where . . . 40
3.11.2 Samostatné vrcholy . . . 40
3.11.3 Vynechání závorek . . . 42
3.12 Celkový pohled na vzory . . . 42

3.13 Výsledné možnosti jazyka 43

4 Ukázkové dotazy **45**
4.1 Výběr dotazů 45
4.2 Dotazy a jejich řešení 45
4.2.1 Výběrové dotazy 45
4.2.2 Extrakční dotazy 48
4.2.3 Redukční dotazy 49
4.2.4 Restrukturalizační dotazy 49
4.2.5 Agregační dotazy 50
4.2.6 Dotazy na kombinování a odvozování 51

5 Praktické srovnání jazyků **55**

6 Pilotní implementace **57**
6.1 Trisolda a dotazy 57
6.2 Úpravy Trisoldy 59
6.3 Implementace překladače 60
6.4 Omezení pilotní implementace 61

7 Závěr **62**

A Agregační funkce **63**

B Základní podpora pro RDF Schema **67**

C Gramatika **69**

D Obsah CD **71**

Příklady

2.1 Reifikace v jazyce SeRQL 5
2.2 Reifikace v jazyce SeRQL bez využití podpory v syntaxi 6
2.3 Vzor trojice v jazyce SPARQL 9
2.4 Vzor trojice v jazyce SeRQL 9
2.5 Vzor trojice v jazyce TRIPLE 9
2.6 Zkrácený zápis vzoru v jazyce SPARQL 10
2.7 Nezkrácený zápis vzoru v jazyce SPARQL ekvivalentní vzoru z příkladu 2.6 10
2.8 Cesta v jazyce SeRQL 10
2.9 Vzor v jazyce SeRQL ekvivalentní cestě z příkladu 2.8 10
2.10 Cesta v jazyce TRIPLE 10
2.11 Vzor v jazyce TRIPLE ekvivalentní cestě z příkladu 2.10 10
2.12 Větvení v jazyce SeRQL 10
2.13 Cesty v jazyce SeRQL ekvivalentní větvení z příkladu 2.12 10
2.14 Větvení v jazyce TRIPLE 11
2.15 Cesty v jazyce TRIPLE ekvivalentní větvení z příkladu 2.14 11
2.16 Vzor anonymního uzlu 11
2.17 Vzor s anonymním uzlem ekvivalentní příkladu 2.16 12
2.18 Cesta v jazyce SPARQL 12
2.19 Vzor v jazyce SPARQL ekvivalentní příkladu 2.18 12
2.20 Vzor grafu v jazyce SeRQL 12
2.21 Vzor grafu v jazyce SeRQL ekvivalentní příkladu 2.20 12
2.22 Cesta v jazyce Versa 13
2.23 Iterace v jazyce XsRQL 13
2.24 Cesta v jazyce XsRQL 14
2.25 Výběr nepovinných dat v jazyce SPARQL 14
2.26 Výběr nepovinných dat v jazyce SeRQL 14
2.27 Výběr nepovinných dat v jazyce Versa 15
2.28 Výběr nepovinných dat v jazyce XsRQL 15
2.29 Vytvoření jmen zaměstnanců 16
2.30 Možný výsledek vytváření jmen zaměstnanců 17
2.31 Vytvoření množiny zaměstnanců 17
2.32 „Chybný“výsledek vytváření množiny zaměstnanců 17
2.33 Výběr prvků seznamu v jazyce Versa 18

2.34 Výběr prvků seznamu v jazyce ARQ 19
2.35 Popis odvozování rdfs: slovníku v jazyce TRIPLE 21
2.36 Použití odvozování v jazyce TRIPLE 21
3.1 Jednoduchý dotaz v jazyce Tequila 26
3.2 Výběr seznamu v jazyce Tequila 27
3.3 Vstupní data příkladu 3.2 29
3.4 Vytvoření trojice 33
3.5 Vytvoření inverzního predikátu 33
3.6 Vylepšené vytvoření inverzního predikátu 34
3.7 Vytvoření strukturovaných dat – první možnost 34
3.8 Vytvoření strukturovaných dat – druhá možnost 35
3.9 Vytvoření strukturovaných dat – správná možnost 35
3.10 Samotné použití vytvoření strukturovaných dat 35
3.11 Použití vzoru from 36
3.12 Vytvoření jmen zaměstnanců 39
3.13 Vytvoření množiny zaměstnanců 39
3.14 Konverze kontejneru 41
3.15 Použití konverze kontejneru 41
3.16 Konverze kontejneru na seznam 44
4.1 Schéma ukázkových dat 46
4.2 Ukázková data 47
4.3 Řešení dotazu č. 1 v jazyce Tequila 47
4.4 Řešení dotazu č. 2 v jazyce Tequila 48
4.5 Řešení dotazu č. 3 v jazyce Tequila 49
4.6 Vyjádření rozdílu dotazů v jazyce Tequila 49
4.7 Řešení dotazu č. 4 v jazyce Tequila 50
4.8 Řešení dotazu č. 5 v jazyce Tequila 50
4.9 Řešení dotazu č. 6 v jazyce Tequila 52
4.10 Řešení dotazu č. 7 v jazyce Tequila 53
4.11 Řešení dotazu č. 8 v jazyce Tequila 54
4.12 Řešení dotazu č. 9 v jazyce Tequila 54
A.1 Implementace pojmenovaného vzoru `tql:max` 64
A.2 Implementace pojmenovaného vzoru `tql:count` 65
A.3 Implementace pojmenovaného vzoru `tql:sum` 66
A.4 Implementace pojmenovaného vzoru `tql:avg` 66
B.1 Implementace pojmenovaného vzoru `ex:subClassesOf` 68

Název práce: Dotazovací jazyky pro Sémantický web
Autor: Jakub Galgonek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
e-mail vedoucího: jakub.yaghob@mff.cuni.cz

Abstrakt: Myšlenka Sémantického webu přináší potřebu vytvářet a uchovávat metadata o dokumentech či obecně zdrojích a následně také v těchto metadatech vyhledávat. Existující dotazovací jazyky pro Sémantický web jsou však buď příliš slabé a nebo mají složitou syntaxi či sémantiku.
Cílem této práce je srovnat existující dotazovací jazyky pro Sémantický web a navrhnout jazyk vlastní hlavně s přihlédnutím na jeho výrazovou sílu.
Srovnání jazyků je prováděno srovnáváním jejich přístupů k jednotlivým oblastem problematiky dotazování, jakými jsou základní výběr dat, schopnost vybrat data s rekurzivně definovanou strukturou, vytváření dat, způsob práce s anonymními uzly a další.
Na základě tohoto srovnání je navržen jazyk Tequila, který je založen na pojmenovaných vzorech a umožňuje obecnou rekurzi.
Tato práce dále ukazuje způsob použití jazyka Tequila a na praktických příkladech ho srovnává s ostatními dotazovacími jazyky.

Klíčová slova: Sémantický web, dotazovací jazyky, RDF, Tequila, Trisolda

Title: Query languages for the Semantic web
Author: Jakub Galgonek
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.
Supervisor's e-mail address: jakub.yaghob@mff.cuni.cz

Abstract: The idea of the Semantic Web brings new requirements such as to create and store metadata of documents or resources. Also ability to search in such metadata is needed. Existing query languages for the Semantic Web are unfortunately either too weak or have complicated syntax or semantics.
The aim of this thesis is to compare existing Semantic Web query languages and to propose new one considering its expression strength.
This comparison is done by juxtapositioning of their approaches to various issues in querying. Such issues are, for example, a basic selection of data, an ability to select data with recursively defined structure, creating data, a way of working with blank nodes, etc.
On the basis of this comparison, the Tequila language is proposed. The Tequila is based on named pattern and provide general recursion.
This thesis also shows the way how to use Tequila language and further, it compares the Tequila with other query languages.

Keywords: Semantic web, query languages, RDF, Tequila, Trisolda

Kapitola 1

Úvod

Množství dat publikovaných na webu neustále roste. Tento růst má vedle řady výhod i jednu prokazatelnou nevýhodu, a to klesající schopnost najít mezi daty požadované informace. Tento problém se snaží řešit různé vyhledávače, například vyhledávač společnosti Google. Současný Web byl však vytvořen hlavně pro přímé čtení lidmi a s dalším strojovým zpracování publikovaných dat se příliš nepočítalo. Proto tyto vyhledávače fungují převážně na základě vyhledávání slov a frází vyskytujících se v dokumentech a na základě vyhodnocování odkazů mezi dokumenty. Možnosti tohoto způsobu vyhledávání jsou však omezené, navíc fungují dobře jen na dokumenty textové povahy. U obrázků či například hudby jsou možnosti vyhledávání již mnohem horší.

V poslední době je rovněž stále více služeb dostupných prostřednictvím Webu. Tyto služby však buď opět předpokládají přímé použití člověkem nebo pracují s různými vstupními či výstupními formáty dat, takže není možné tyto služby jednoduše propojovat za účelem automatického řešení složitějších úloh.

Odpovědí na tyto problémy by podle Tima Berners-Leeho[16] mělo být rozšíření současného Webu do podoby Sémantického webu. V jím nastíněné představě by Sémantický web měl přinést strukturu do významového obsahu stránek, softwaroví agenti by pak postupovali od stránky ke stránce a řešili netriviální úkoly. Sémantický web předpokládá doplnění existujících dat o metadata, která by pomohla určit jejich význam. Spolu s doplněním metadat je nutné umožnit také v těchto metadatech vyhledávat, což mají umožnit právě dotazovací jazyky pro Sémantický web.

Cílem této diplomové práce je srovnat existující dotazovací jazyky pro Sémantický web hlavně s přihlédnutím na jejich vyjadřovací sílu. Dalším cílem je navrhnout silný, ale zároveň ne příliš složitý, dotazovací jazyk a implementovat ho pro datové úložiště Trisolda[4].

Diplomová práce je rozdělena do sedmi kapitol. Druhá kapitola představuje a porovnává přístupy vybraných dotazovacích jazyků k jednotlivým problémům, které musí dotazovací jazyky řešit. Následující kapitola na základě získaných poznatků navrhuje nový dotazovací jazyk. Čtvrtá kapitola představuje řešení několika ukázkových dotazů v tomto jazyce. Další kapitola pak poskytuje srovnání vybraných dotazovacích jazyků na základě jejich schopnosti vyřešit ukázkové dotazy čtvrté kapitoly. Šestá kapitola stručně popisuje pilotní implementaci překladače navrženého dotazovacího jazyka pro datové úložiště Trisolda.

Kapitola 2

Dotazovací jazyky pro Sémantický web

Tato kapitola provádí stručný úvod do jazyka pro zápis metadat. Vybírá dotazovací jazyky, kterými se bude zabývat, a provádí jejich srovnání na základě jejich přístupu k jednotlivým oblastem problematiky dotazování v prostředí Sémantického webu.

2.1 Jazyk metadat

Pro zápis metadat v prostředí Sémantického webu existuje několik různých jazyků. Tato diplomová práce se omezuje výhradně na jazyk RDF (*Resource Description Framework*) [7][11][10][8][9], který je prosazován W3C konzorciem a který používá také datové úložiště Trisolda.

Metadata jsou v RDF zapisována pomocí trojic (*triple*), skládajících se ze subjektu (*subject*), predikátu (*predicate*) a objektu (*object*). Predikát se někdy nazývá také vlastnost (*property*). Trojice vyjadřuje jednoduché tvrzení, že daný subjekt má vlastnost, jejíž hodnota je objekt. Subjektem může být URI reference (*Uniform Resource Identifier reference*)[15] nebo anonymní uzel (*blank node*), predikátem pouze URI reference a objektem URI reference, anonymní uzel nebo literál. Literály jsou řetězce, které mohou volitelně navíc obsahovat URI datatypu (*datatype URI*) nebo specifikaci jazyka (*language tag*). Na množinu trojic se pohlíží jako na orientovaný graf (*RDF graph*), kde subjekty a objekty představují uzly, predikáty pak orientované hrany vedoucí od subjektů k objektům.

Pro zkrácení zápisu URI referencí se využívá kvalifikovaných jmen (*qualified name*) známých z XML. Kvalifikované jméno se skládá z prefixu (zakončeného dvojtečkou) a lokálního jména. Prefixům bývá přidělena URI reference. Kvalifikované jméno je ekvivalentní URI referenci získané zřetězením URI reference prefixu a lokálního jména. Prefixy používané v této práci včetně jim přiřazených URI referencí shrnuje tabulka 2.1. Prefix `tql:` je využíván nově navrženým dotazovacím jazykem. A prefix `ex:` slouží k označení URI referencí používaných pouze v ukázkových příkladech.

Prefix	URI reference prefixu
rdf:	<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
rdfs:	<http://www.w3.org/2000/01/rdf-schema#>
xsd:	<http://www.w3.org/2001/XMLSchema#>
tql:	<http://ulita.ms.mff.cuni.cz/tequila/term#>
ex:	<http://www.example.org/term#>

Tabulka 2.1: Definice prefixů kvalifikovaných jmen

2.2 Výběr dotazovacích jazyků

Dotazovacích jazyků pro RDF existuje velké množství a není v možnostech této práce se všemi zabývat. Proto bylo za účelem porovnání vybráno pouze několik jazyků, avšak tak, aby celkově pokryly většinu hlavních myšlenek používaných v dotazovacích jazycích pro RDF. Rovněž bylo snahou vybrat z existujících jazyků ty nejznámější. Při výběru výrazně pomohly práce *RDF Querying: Language Constructs and Evaluation Methods Compared*[19] a *A Comparison of RDF Query Languages*[20].

Tato práce se bude zabývat následujícími jazyky:

- SPARQL[13]
- SeRQL[5]
- ARQ[1][2]
- Versa[22]
- XsRQL[21]
- TRIPLE[24]
- Xcerpt[18][23]

Následující podkapitoly porovnávají tyto jazyky na základě jejich přístupu k jednotlivým oblastem problematiky dotazování a poskytují tak základní přehled hlavních myšlenek a zároveň jejich srovnání.

2.3 Primární účel dotazovacího jazyka

Hlavním úkolem dotazovacího jazyka je umožnit vytvářet dotazy, jejichž vyhodnocením je možné získat požadované informace. Podle vztahu získaných informací ke vstupním datům můžeme rozlišit několik možných přístupů:

1. *Ryze dotazovací jazyky*: Dotaz zapsaný v takovémto jazyce představuje v podstatě pouze kriteria pro výběr ze vstupních dat. Výsledkem takového dotazu je tedy vždy jen nějaká podmnožina vstupních dat a dat obsažených přímo v dotazu.

   Do této kategorie nespadají jen velmi jednoduché jazyky, ale také jazyky, u nichž je toto omezení dáno zvolenou filosofií. Jako je tomu například u logického dotazovacího jazyka TRIPLE.

2. *Dotazovací jazyky s možností výpočtu nových hodnot*: Tyto jazyky dokáží s použitím aritmetiky nebo agregačních funkcí popsat vytvoření nové hodnoty a její vložení do výsledku. Právě ona schopnost vložení vypočtené hodnoty do výsledku je pro zařazení jazyka do této kategorie klíčová, neboť i mnoho ryze dotazovacích jazyků sice obsahuje podporu pro aritmetické výrazy, ty však lze použít pouze v konstrukcích typu `FILTER`.

   Příkladem dotazovacího jazyka s aritmetikou je jazyk Versa.

3. *Dotazovací jazyky s možností výpočtu nových trojic*: Tyto dotazovací jazyky jsou schopny, na rozdíl od předešlé kategorie, popsat nejen vytvoření nových hodnot, ale rovněž nových trojic. Ne vždy se však jedná o uzavřené dotazovací jazyky v tom smyslu, že by výsledek jednoho dotazu šel použít jako vstupní data pro další dotaz. Přitom uzavřenost jazyka je velmi užitečnou vlastností. Umožňuje například provést jedním dotazem transformaci dat z různých databází na data využívající jednotný slovník a takto transformovaná data použít jako vstupní data pro jiný dotaz, který se tak nebude muset zabývat rozdílnou povahou vstupních dat.

   Je dobré se na tomto místě zmínit, že výše zmíněné výhody lze dosáhnout i bez nutnosti explicitního vytvoření RDF grafu nějakým dotazem. Například jazyk TRIPLE definuje model jako množinu trojic a pravidel, popisujících jak z existence určitých trojic odvodit trojice nové. Model je možné parametrizovat jiným modelem. Parametrizovaný model pak v podstatě provádí transformaci zadaného modelu. Dotaz, který by se v uzavřeném jazyce zapsal jako dotaz nad grafem explicitně vytvořeným ze vstupního grafu nějakým poddotazem, se v jazyce TRIPLE ekvivalentně zapíše jako dotaz nad modelem, který je parametrizovaný vstupním modelem.

   Do této kategorie patří například jazyky XsRQL, ARQ či Xcerpt, přičemž jen jazyk Xcerpt lze považovat za uzavřený.

Toto dělení samozřejmě není nikterak přesné, pouze se snaží ukázat, že k návrhu dotazovacího jazyka lze přistupovat s různými cíli. Některé jazyky nemusí být možné přesně zařadit do některé z výše zmíněných kategorií. Například jazyky SPARQL nebo SeRQL umožňují vytvářet nové trojice, ale vždy jen s použitím původních hodnot. Jazyk SPARQL navíc umožňuje vytvořit RDF graf až v poslední fázi dotazu a již s ním neumí dále pracovat.

Nejsilnější výrazové prostředky nabízí obecně jazyky z poslední kategorie, zvláště pokud jsou uzavřené. Často je to však na úkor jednoduchosti takového jazyka. Pokud má jazyk sloužit pouze k získání odpovědi (a na jejím formátu příliš nezáleží), může být jazyk pouze s aritmetikou dobrým kompromisem.

2.4 Podpora slovníků

Další odlišnosti v základním přístupu dotazovacích jazyků lze pozorovat u podpory slovníků. Lze rozlišit tyto možné přístupy:

1. *Jazyky bez podpory*: Nejjednodušší jazyky nepodporují žádný slovník a ani nenabízí obecné prostředky, jak podporovat libovolný slovník. V těchto jazycích jsou si všechny URI reference rovnocenné a žádná nemá speciální význam. Do této skupiny patří například jazyk SPARQL.

2. *Jazyky s vestavěnou podporou*: V tomto směru dál jsou na tom jazyky, které mají zabudovánu podporu pro některé slovníky, nejčastěji pro rdf: slovník a rdfs: slovník. Této podpory může být dosaženo různými způsoby:

   (a) *Vestavěné odvozování*: Nejjednodušší podpora slovníku je pomocí vestavění jeho odvozování (*entailment*). Tento druh podpory je možné snadno přidat do již existujících jazyků. Specifikace jazyka SPARQL dokonce na toto pamatuje a popisuje, jaké podmínky je třeba při použití jiných způsobů odvozování dodržet.

   Vestavěným odvozováním rdfs: slovníku disponuje například jazyk SeRQL.

   (b) *Vestavěné funkce a dotazy*: O něco lepší podpory lze dosáhnout přidáním vestavěných funkcí a dotazů.

   Například jazyk Versa obsahuje funkci `type`, jejímž parametrem je jméno třídy a která vrací všechny instance dané třídy.

   (c) *Na úrovni syntaxe*: Nejlepší podpory pro slovník je možné dosáhnout jeho podporou už na úrovni syntaxe jazyka. S takovouto podporou je nutné počítat již při návrhu jazyka.

   Příkladem může být podpora reifikace v jazyce SeRQL, kdy je možné reifikaci trojice provést jejím zápisem do složených závorek, jak je vidět na příkladu 2.1, který je, za předpokladu, že reifikované trojici byl přiřazen anonymní uzel s interním jménem `_:blank`, ekvivalentní příkladu 2.2.

3. *Jazyky s obecnou podporou*: Některé jazyky nabízejí dost silné výrazové prostřekdy, které jim umožní pracovat s daným slovníkem bez vestavěné podpory pro tento slovník. Mezi takové jazyky patří například jazyk TRIPLE nebo Xcerpt, umožňující snadno popsat libovolné odvozovací pravidla a vyjádřit tak odvození (*entailment*) slovníku.

   Odvození však ne vždy stačí k efektivní práci se slovníkem, neumožňuje například popsat přidání prvku do seznamu a podobné činnosti.

```
{ {reifSubject} reifPredicate {reifObject} }
```

Příklad 2.1: Reifikace v jazyce SeRQL

Není asi překvapující, že nejlepší a nejúplnější podpory pro slovník je dosaženo při jejím vestavění přímo do jazyka, zvláště když je tak učiněno již na úrovni syntaxe. Myšlenka Sémantického webu však předpokládá kombinování informací z různých RDF databází, které obecně mohou používat různé slovníky, a to například včetně slovníků pro popis ontologií. Vestavěná podpora pro velké množství používaných slovníků samozřejmě není zrovna dobrým řešením.

Jako nejlepší řešení se jeví možnost opatřit jazyk dostatečně silnými výrazovými prostředky pro obecnou podporu slovníků. A dále umožnit tuto podporu ukládat do externích modulů. Pokud totiž se silnými prostředky nenabízí jazyk také nějakou formu abstrakce, umožňující si například pojmenovat často používané poddotazy, může být takováto práce se slovníkem velmi nepohodlná.

Řešením by tedy mohlo být vytvoření obecného snadno rozšiřitelného dotazovacího jazyka.

2.5 Původ jazyků

Většina dnešních dotazovacích jazyků pro RDF má svůj původ, byť nepřímo, ve starších jazycích, které byly původně navrženy pro jiné účely. Důvody pro to jsou obecně asi dvojí. Prvním důvodem je snaha o použití již osvědčeného přístupu. Druhým a asi hlavním důvodem však je snaha usnadnit lidem přístup k RDF jazykem, který je podobný jazyku, který již ovládají. Tato snaha o podobnost sice na jedné straně umožňuje lidem (za předpokladu, že ovládají původní jazyk) rychle si osvojit také nový jazyk, na druhé straně však často vede k tomu, že výrazové prostředky nového jazyka jsou slabší, neboť převzatá filosofie se na dotazování RDF dat příliš nehodí.

V prostředí RDF dotazovacích jazyků lze rozlišit tyto proudy:

1. *Jazyky odvozené od jazyka SQL*: Asi nejznámějším a nejpoužívanějším dotazovacím jazykem dneška je jazyk SQL. Nepřekvapí tedy, že nejznámější dotazovací jazyky pro RDF v mnohém vychází právě z tohoto jazyka.

   Inspirace jazykem SQL je dobře vidět například na jazyce SeRQL. Klauzule `FROM`, která u jazyka SQL popisuje spojení tabulek, obsahuje u jazyka SeRQL vzor, na jehož základě jsou ve formě tabulky vybrána data z RDF databáze. Význam ostatních klauzulí jazyka SeRQL se pak už příliš neliší od jejich významu v jazyce SQL.

```
{_:blank} rdf:type {rdf:Statement},
{_:blank} rdf:subject {reifSubject},
{_:blank} rdf:predicate {reifPredicate},
{_:blank} rdf:object {reifObject}
```

Příklad 2.2: Reifikace v jazyce SeRQL bez využití podpory v syntaxi

Dalším jazykem inspirovaným nepřímo jazykem SQL je jazyk SPARQL. Jeho syntaxe je už od syntaxe jazyka SQL vzdálenější, přesto sémantika tohoto jazyka je stále těsně spjata s relačním kalkulem. Jazykem SPARQL je pak inspirován jazyk ARQ.

Přestože se těmto jazykům dostává mnoho pozornosti, pro semistrukturovaná data uložená v RDF jsou prostředky jazyků odvozených od SQL často nedostačující.

2. *Jazyky odvozené od dotazovacích jazyků pro XML*: Vedle RDF nachází v Sématickém webu velkého uplatnění také XML, navíc datový model XML (stromy s pojmenovanými vrcholy) je mnohem bližší RDF než relační datový model. Přizpůsobení existujících dotazovacích jazyků pro XML pro účely dotazování RDF se tedy také nabízí.

   Do kategorie těchto jazyků patří jazyk XsRQL, inspirovaný jazykem XQuery.

3. *Funkcionální programovaní*: Inspirace jinými funkcionálními přístupy již tak běžné nejsou. Zástupcem této kategorie je například jazyk Versa, inspirovaný jazykem Lisp.

   Základní datovou strukturou tohoto jazyka je (stejně jako v jazyce Lisp) seznam. Velmi snadno lze po grafu přecházet od nějakého seznamu uzlů k jinému. Avšak v případech, kdy je třeba vrátit nějakou komplexnější odpověď, je již použití seznamů velmi nepohodlné.

4. *Logické programování*: Zajímavé je použití myšlenek logického programování. Zástupcem takového dotazovacího jazyka je jazyk TRIPLE, který je syntaktickým rozšířením Hornovy logiky.

   Jednotlivé trojice představují fakta, dále je možné definovat pravidla. Vyhodnocování dotazu, který má tvar formule, pak odpovídá hledání takového ohodnocení proměnných, pro které je formule pravdivá.

   V takovémto přístupu k dotazování se velmi snadno pracuje s odvozováním (*entailment*). Horší už je to však s možností podpory aritmetiky či agregačních funkcí.

5. *Grafové dotazovací jazyky*: Grafové dotazovací jazyky většinou používají dost silný datový model, takže je možné použít je bez úprav i na dotazování RDF dat. Jedinou podmínkou je definovat mapování z RDF na jejich datový model.

   Příkladem takového jazyka je jazyk Xcerpt, který byl původně určen hlavně na dotazování XML, ale například také HTML.

   Datový model jazyka Xcerpt je v základu strom, který však může obsahovat transparentní reference, takže je možné popsat v podstatě libovolný souvislý graf. Mapování z RDF na datové termy jazyka Xcerpt bylo navrženo dvojí[17]:

   (a) *Mapování jako graf*: Grafy používané v jazyce Xcerpt nemají pojmenované hrany, proto je při mapování každá hrana z RDF grafu nahrazena stejně pojmenovaným uzlem ležícím v Xcerpt grafu na cestě mezi původními uzly. Jenot-

livé komponenty souvislosti jsou pak spojeny s „fiktivním“ uzlem `RDF`, aby byla zajištěna požadovaná souvislost výsledného grafu.

Při dotazování nad takovým grafem je možné přímo využít všech možností jazyka Xcerpt. Převodem je však ztracena informace o tom, který uzel představoval subjekt či objekt a který predikát.

(b) *Mapování po trojicích*: Druhou možností je mapovat každou z trojic zvlášť, pomocí elementu `RDF-TRIPLE`, obsahující podelementy představující subjekt, predikát a objekt. Tím je zachována veškerá původní informace. Cenou za to je však nemožnost přímého využití například takových možností jazyka Xcerpt, jakými je dotaz zadaný pomocí (rekurzivní) cesty grafem.

Převod RDF grafu do jiného datového modelu a psaní dotazů v jazyce pro tento model je samozřejmě nepřímočaré a nepohodlné. Ovšem použití jednoho dotazovacího jazyka jak pro RDF, tak pro XML s sebou přináší také několik výhod. Dotazovací jazyk tak například může klást dotazy na obsah literálů typu `rdf:XMLLiteral`. Naopak je zase možné při hledání informací v XML dokumentu, který je opatřen odkazy na RDF data popisující sémantiku elementů daného XML dokumentu, volně přejít k těmto datům a vyvodit z informací v nich obsažených nějaké závěry o obsahu zkoumaného XML dokumentu.

2.6 Výběr dat z RDF databáze

Základní výběr dat z RDF databáze je bezpochyby jednou z nejdůležitějších konstrukcí dotazovacího jazyka. Bez ohledu na celkovou filosofii jazyka lze v tomto směru rozlišit dva hlavní proudy:

1. *Výběr pomocí vzorů*: Tento typ výběru dat funguje obecně tak, že v RDF databázi jsou hledány podgrafy, které odpovídají danému vzoru. Každý z jazyků spadajících do této kategorie dokáže zapsat jednoduchý vzor na trojici (*triple pattern*). Vzor trojice může, na rozdíl od datové trojice, obsahovat navíc proměnné, a to na libovolné pozici. Množina vzorů trojic pak tvoří samotný vzor. Při vyhodnocování vzoru se hledá takové ohodnocení proměnných, pro které vstupní data obsahují jako podgraf daný vzor.

   Podmínky na hodnoty se v těchto jazycích vyjadřují většinou mimo vlastní vzor pomocí konstrukcí typu `FILTER` (v případě jazyka SPARQL) nebo pomocí speciálních klauzulí (například klauzule `WHERE` v případě jazyka SeRQL).

   Příklad 2.3 ukazuje vzor trojice zapsaný v jazyce SPARQL, příklad 2.4 v jazyce SeRQL a příklad 2.5 v jazyce TRIPLE. Vzory jazyka ARQ odpovídají v základu vzorům jazyka SPARQL. Také jazyk Xcerpt využívá vzorů, protože se však jedná o vzory pro jeho vlastní datový model, nebudeme se jimi v této práci zabývat.

```
?subject ?predicate ?object.
```

Příklad 2.3: Vzor trojice v jazyce SPARQL

```
{subject} predicate {object}
```

Příklad 2.4: Vzor trojice v jazyce SeRQL

Jazyky většinou obsahují různý syntaktický cukr umožňující zápis vzoru zkrátit. Přístup k tomuto je obecně dvojí:

(a) *Grafové vzory*: Tento přístup se zaměřuje na omezením nutnosti psát ve vzorech trojic subjekt a nebo subjekt a predikát, pokud jsou stejné jako u předešlého vzoru trojice.

Například v jazyce SPARQL se vzory trojic se stejným subjektem oddělují středníkem, přičemž subjekt se píše pouze u prvního z nich. Podobně se vzory trojic se stejným subjektem a predikátem oddělují čárkou. Takovýto zkrácený zápis je možné vidět na příkladu 2.6, který je ekvivalentní příkladu 2.7 napsaném v nezkrácené formě.

(b) *Cesty*: Odlišný přístup umožňuje spojovat vzory trojic, kde objekt jednoho odpovídá subjektu druhého, a vytvářet tak delší cesty (*path expressions*). Celkový vzor je pak množinou takovýchto cest.

Příklad 2.8 ukazuje zápis cesty v jazyce SeRQL, která je ekvivalentní vzorům trojic uvedeným v příkladu 2.9, příklad 2.10 pak zápis cesty v jazyce TRIPLE ekvivalentní vzorům trojic z příkladu 2.11.

Jazyky zaměřené na cesty umožňují často také tyto cesty větvit. V jazyce SeRQL se cesta větví pomocí středníku, a to tak, že větev následující za středníkem přebírá jako svůj první subjekt poslední subjekt před středníkem. Možnosti takového způsobu větvení jsou však velmi omezené, neboť v popisu první větve není možné dále pokračovat, viz příklad 2.12, který je ekvivalentní příkladu 2.13.

Jazyk TRIPLE také umožňuje větvení pomocí středníku. Díky povinnému uzávorkování predikátu a objektu však stejným nedostatkem netrpí, jak ukazuje příklad 2.14, který je ekvivalentní příkladu 2.15.

Výše zmíněné rozdělení samozřejmě není nijak striktní. Například jazyk SPARQL obsahuje možnost popsat anonymní uzel bez nutnosti použít jeho interní jméno. Anonymní uzel se popíše použitím hranatých závorek, do nichž je možné zapsat jeho

```
subject[predicate->object]
```

Příklad 2.5: Vzor trojice v jazyce TRIPLE

```
?subject ?predicate_1 ?object_1,
                      ?object_2;
         ?predicate_3 ?object_3,
                      ?object_4.
```

Příklad 2.6: Zkrácený zápis vzoru v jazyce SPARQL

```
?subject ?predicate_1 ?object_1.
?subject ?predicate_1 ?object_2.
?subject ?predicate_3 ?object_3.
?subject ?predicate_3 ?object_4.
```

Příklad 2.7: Nezkrácený zápis vzoru v jazyce SPARQL ekvivalentní vzoru z příkladu 2.6

```
{subject} predicate_1 {node} predicate_2 {object}
```

Příklad 2.8: Cesta v jazyce SeRQL

```
{subject} predicate_1 {node},
{node} predicate_2 {object}
```

Příklad 2.9: Vzor v jazyce SeRQL ekvivalentní cestě z příkladu 2.8

```
subject[predicate_1->node[predicate_2, object]]
```

Příklad 2.10: Cesta v jazyce TRIPLE

```
subject[predicate_1->node]
AND
node[predicate_2->object]
```

Příklad 2.11: Vzor v jazyce TRIPLE ekvivalentní cestě z příkladu 2.10

```
{subject} predicate {node} predicate_1 {object_1};
                           predicate_2 {node_2} predicate_3 {object_2}
```

Příklad 2.12: Větvení v jazyce SeRQL

```
{subject} predicate {node} predicate_1 {object_1},
{node} predicate_2 {node_2} predicate_3 {object_3}
```

Příklad 2.13: Cesty v jazyce SeRQL ekvivalentní větvení z příkladu 2.12

```
subject[predicate->node[predicate_1->node_1[predicate_2->object_2];
                        predicate_3->node_3[predicate_4->object_4]]]
```

Příklad 2.14: Větvení v jazyce TRIPLE

```
subject[predicate->node[predicate_1->node_1[predicate_2->object_2]]]
AND
node[predicate_3->node_3[predicate_4->object_4]]
```

Příklad 2.15: Cesty v jazyce TRIPLE ekvivalentní větvení z příkladu 2.14

vlastnosti s jejich hodnotami. Použití této konstrukce ukazuje příklad 2.16, který je za předpokladu, že popisovanému anonymnímu uzlu bylo přiřazeno jméno `_:blank`, ekvivalentní příkladu 2.17.

Tuto konstrukci lze použít přímo ve vzoru trojice, čímž je možné vytvořit obdobu cesty, ovšem s vnitřními uzly cesty omezenými na anonymní uzly, jak ukazuje příklad 2.18, který je ekvivalentní příkladu 2.19.

Dále například jazyk SeRQL umožňuje při specifikaci uzlu nebo predikátu uvést rovnou celý seznam přípustných hodnot oddělených čárkou. Taková cesta je pak ekvivalentní množině cest, u kterých z každého seznamu přípustných hodnot byla vybrána vždy právě jedna varianta. Pokud navíc seznam obsahoval více proměnných, je implicitně doplněna podmínka na různost hodnot těchto proměnných. Tímto způsobem lze tedy v jazyce v SeRQL omezit nutnost opakovat stejný subjekt a predikát.

2. *Navigační přístup*: Druhým obecným přístupem je navigační přístup, který obecně popisuje cestu, kterou se z nějaké vstupní množiny uzlů lze dostat k výstupní množině uzlů.

   V jazyce Versa toto zajišťuje konstrukce, které se říká *forward traversal*. Ta má obecně tvar:

   ```
   subj_list_exp - pred_list_exp -> boolean_exp
   ```

   První list se vyhodnotí a poskytne množinu subjektů. Pro každý subjekt, který se použije jako kontext[1], se vyhodnotí druhý list představující množinu možných predikátů. Následně se v RDF databázi vyhledají trojice s odpovídajícím subjektem a predikátem. Do výsledku se pak zahrnou ty objekty, pro které výraz `boolean_exp`, s daným objektem jako kontextem, nabývá hodnoty `true`.

[1] Kontext představuje v jazyce Versa způsob, jak předat výrazu parametr.

```
[ ?predicate_1 ?object_1; ?predicate_2 ?object_2 ]
```

Příklad 2.16: Vzor anonymního uzlu

```
_:blank ?predicate_1 ?object_1;
        ?predicate_2 ?object_2.
```

Příklad 2.17: Vzor s anonymním uzlem ekvivalentní příkladu 2.16

```
?subject ?predicate [ ?predicate_1 [ ?predicate_2 ?object_2 ];
                      ?predicate_3 [ ?predicate_4 ?object_4 ]]
```

Příklad 2.18: Cesta v jazyce SPARQL

```
?subject ?predicate _:blank1.
_:blank1 ?predicate_1 _:blank2.
_:blank2 ?predicate_2 ?object_2.
_:blank1 ?predicate_3 _:blank3.
_:blank3 ?predicate_4 ?object_4.
```

Příklad 2.19: Vzor v jazyce SPARQL ekvivalentní příkladu 2.18

```
{subject} predicate {object_1, object_2, object_3}
```

Příklad 2.20: Vzor grafu v jazyce SeRQL

```
FROM
  {subject} predicate {object_1},
  {subject} predicate {object_2},
  {subject} predicate {object_3}
WHERE object_1 != object_2 AND
      object_1 != object_3 AND
      object_2 != object_3
```

Příklad 2.21: Vzor grafu v jazyce SeRQL ekvivalentní příkladu 2.20

```
(rdfs:Class <- rdf:type - true) - rdfs:label -> true
```

Příklad 2.22: Cesta v jazyce Versa

```
for $x in *[ @<marriedTo> ]
$x/@<marriedTo>/*
```

Příklad 2.23: Iterace v jazyce XsRQL

Naprosto stejně se vyhodnocuje i *forward filtering*, který má obecně tuto podobu:

```
subj_list_exp |- pred_list_exp -> boolean_exp
```

Rozdíl je pouze v tom, že do výsledku se zahrnují z vybrané trojice subjekty místo objektů.

Přesně naopak než forward traversal se vyhodnocuje *backward traversal*, který má obecně tuto podobu:

```
obj_list_exp <- pred_list_exp - boolean_exp
```

První list se vyhodnotí a poskytne množinu objektů. Pro každý objekt, který se použije jako kontext, se vyhodnotí druhý list představující množinu možných predikátů. Následně se v RDF databázi vyhledají trojice s odpovídajícím objektem a predikátem. Do výsledku se pak zahrnou ty subjekty, pro které výraz `boolean_exp`, s daným subjektem jako kontextem, nabývá hodnoty `true`.

Tyto výrazy lze do sebe zanořovat a vytvářet tak delší cesty (viz příklad 2.22).

Další jazyk využívající navigační přístup je jazyk XsRQL. Ten umožňuje popsat cestu, která se pak hledá v RDF databázi. Na rozdíl od cest popsaných u jazyků založených na vzorech, tato cesta neobsahuje proměnné (ve smyslu proměnných vzoru) a výsledkem je vždy množina uzlů, přes kterou jde pomocí proměnné iterovat, jak ukazuje příklad 2.23.

Jednotlivé části cesty (tj. uzly a predikáty) se v jazyce XsRQL oddělují lomítkem. Jména predikátů se uvozují znakem `@`, čímž se odlišují od jmen uzlů. Jako jméno lze kromě konstanty použít také proměnnou, té ale musí být přiřazena hodnota ještě před vyhodnocováním cesty. Jako jméno uzlu (resp. predikátu) je možné použít také hvězdičku, pak jméno uzlu (resp. predikátu) není podstatné. Podmínky na predikáty (a jejich hodnoty) se píšou za jméno uzlu do hranatých závorek. Příklad 2.24 ukazuje cestu, jejímž výsledkem jsou všechny objekty v RDF databázi.

Vzory představují obecně lepší způsob výběru dat, neboť mohou snadněji popsat data se složitější strukturou. Navíc se v nich snadněji vyjadřují hodnotové podmínky, pokud tyto závisí na hodnotách z více uzlů, neboť při navigačním přístupu se podmínka na hodnotu uzlu píše přímo k tomuto uzlu. Jazyk XsRQL, mající proměnné, může tento nedostatek

```
@*/*
```

Příklad 2.24: Cesta v jazyce XsRQL

```
?person rdf:type ex:Person.
?person ex:name ?name.

optional
{
    ?person ex:email ?email.
}
```

Příklad 2.25: Výběr nepovinných dat v jazyce SPARQL

celkem snadno obejít postupnou iterací přes několik proměnných a jejich použitím v podmínkách. Ale například jazyk Versa takovou možnost nemá a vyjádření takových dotazů je v něm náročnější a neintuitivní.

2.7 Nepovinná data

Informace uložené v RDF nemusí být vždy kompletní. Proto je vhodné mít možnost v jazyce snadno vyjádřit, že některá data nemusí být pro úspěšné vykonání dotazu v RDF databázi přítomna, ale pokud tam přítomna jsou, mají být do odpovědi zahrnuta. Příkladem může být dotaz, který má vypsat jména všech osob (zdrojů typu `ex:Person`) a u těch, které mají email, také jejich email.

V jazycích využívajících vzory se takovéto dotazy vyjádří velmi jednoduše tak, že se část vzoru označí za nepovinnou. V jazyce SPARQL nebo ARQ se takováto část vzoru uzavře do bloku uvozeného klíčovým slovem `optional` (viz příklad 2.25), podobně v jazyce Xcerpt. V jazyce SeRQL se nepovinná část cesty uzavře do hranatých závorek (viz příklad 2.26).

U jazyků využívajících navigační přístup nemá označení části cesty za nepovinnou dobrý smysl. Výběr nepovinných dat je tedy nutné například spojit s testem na existenci těchto dat, jak se to provádí v jazyce XsRQL (viz příklad 2.28). Jiný přístup používá například jazyk Versa, kde se napřed získá seznam osob, které mají také jméno, a následně se pro každou osobu na seznamu získají všechna její jména a všechny (nebo i žádný) její emaily (viz příklad 2.27).

I v této oblasti tedy jazyky s navigačním přístupem zaostávají.

```
{Person} rdf:type ex:Person;
         ex:name  {Name};
        [ex:email {EmailAddress}]
```

Příklad 2.26: Výběr nepovinných dat v jazyce SeRQL

```
distribute(type(ex:Person) :- ex:name -> true,
        "all() - ex:name -> true",
        "all() - ex:mail -> true")
```

Příklad 2.27: Výběr nepovinných dat v jazyce Versa

```
for $person in *[ @rdf:type = ex:Person and @ex:name]
return
    { $person, @ex:name, $person/@ex:name/* },
    if ( exists( $person/@ex:mail ))
    then { $person, @ex:mail, $person/@ex:mail/* }
    else ()
```

Příklad 2.28: Výběr nepovinných dat v jazyce XsRQL

2.8 Přístup k anonymnímu uzlu

Některé jazyky umožňují přímo v dotazu zapsat identifikaci anonymního uzlu. Časté je to u jazyků používajících pro výběr dat vzory, neboť vzory často vychází z nějakého formátu pro serializaci RDF dat, přičemž tyto formáty potřebují vnitřně identifikovat anonymní uzel, a proto obsahují syntax pro zápis jeho interního jména (nejčastěji ve formě kvalifikovaného jména s prefixem ve tvaru `_:`). Jazyky tuto schopnost často přejímají, liší se však sémantikou takto zapsaného anonymního uzlu.

Mezi možné přístupy patří:

1. *Existenční kvalifikace*: Použití anonymního uzlu ve vzoru jazyka SPARQL má význam existenční kvalifikace, říkající pouze „existuje uzel, takový že . . . “. Anonymní uzly se tedy v jazyce SPARQL chovají jako proměnné, jenž se po vyhodnocení vzoru odstraní projekcí.

   Toto chování je dáno tím, že při vyhodnocování vzoru jazyka SPARQL se nehledá takové ohodnocení proměnných, pro které se vzor přímo nachází ve vstupních datech, ale hledá se takové ohodnocení, pro které je možné ze vstupních dat jednoduše odvodit (*simple entailment*) graf, který je, pro dané ohodnocení proměnných, ekvivalentní vzoru.

2. *Identifikace*: V jazyce SeRQL má zápis anonymního uzlu význam konkrétní identifikace anonymního uzlu v databázi. Jazyk SeRQL tedy porušuje představu anonymních uzlů jako nepojmenovaných uzlů a umožňuje v dotazech používat jejich interní jména. Pokud je třeba v jazyce SeRQL vyjádřit existenční kvalifikace uzlu, lze tak učinit pomocí prázdných složených závorek.

Přístup jazyka SeRQL vychází pravděpodobně ze snahy umožnit použití identifikace anonymního uzlu vrácené jako výsledek jednoho dotazu v dotazu dalším. To samo o sobě

```
CONSTRUCT
{
    ?person ex:name _:blank.
    _:blank ex:first ?first_name.
    _:blank ex:surname ?surname.
}
WHERE
{
    # Výběr ?first_name a ?surname
}
```

Příklad 2.29: Vytvoření jmen zaměstnanců

není špatná věc, pokud by se ovšem toto předání dělo například pomocí ohodnocení proměnné a tedy způsobem, kde by konkrétní identifikace anonymního uzlu, použitá v dané databázi, zůstala před uživatelem skryta. Jazyk SeRQL však není uzavřený a tento způsob použít nemůže.

2.9 Vytváření RDF dat

Vytváření RDF grafu lze provést několika způsoby:

1. *Pomocí vzoru*: Tento přístup k vytváření RDF grafu používají snad všechny jazyky (pokud vytváření umožňují), jejichž výběr dat je založen na vzorech. Z porovnávaných jazyků jsou to jazyky SPARQL, SeRQL, ARQ a Xcerpt.

   Vytváření RDF grafu zde tak probíhá zcela přirozeně opět pomocí vzoru, do kterého se za proměnné v něm obsažené postupně dosazují hodnoty z ohodnocení proměnných získané z dotazovací části. Grafy získané pro jednotlivá ohodnocení se pak spojují do výsledného grafu.

   Jediným úskalím tohoto přístupu jsou anonymní uzly a otázka, jestli se před spojením grafů mají anonymní uzly přejmenovat tak, aby různé grafy neměly anonymní uzly stejného (interního) jména. Například v jazyce SPARQL se anonymní uzly přejmenovávají.

   Uvažujme příklad, kdy chceme osobám přidělit vlastnost `ex:name` reprezentující jméno, které se skládá z křestního jména (vlastnost jména `ex:first_name`) a příjmení (vlastnost jména `ex:surname`).

   Při použití jazyka SPARQL (viz příklad 2.29), je získán očekávaný výsledek, který může vypadat například tak, jako v příkladu 2.30.

   Pokud bychom však například chtěli jednotlivým oddělením ve firmě přes vlastnost `ex:persons` připojit anonymní uzel, ke kterému jsou přes vlastnost `ex:in` připojeni zaměstnanci, narazíme v jazyce SPARQL na problém, neboť požadujeme, aby

```
ex:person_01 ex:name _:blank1.
_:blank1 ex:first "Jakub".
_:blank1 ex:surname "Galgonek".

ex:person_01 ex:name _:blank2.
_:blank2 ex:first "Josef".
_:blank2 ex:surname "Novak".
```

Příklad 2.30: Možný výsledek vytváření jmen zaměstnanců

```
CONSTRUCT
{
    ?department ex:persons _:blank.
    _:blank ex:in ?person.
}
WHERE
{
    # Výběr ?department a ?person
}
```

Příklad 2.31: Vytvoření množiny zaměstnanců

každé oddělení mělo přes vlastnost `ex:persons` připojen právě jeden anonymní uzel. Tento problém demonstruje příklad 2.31 a jeho možný „chybný“ výsledek v příkladu 2.32). Tento příklad zároveň ukazuje, že řešením by nebylo rozdělení anonymních uzlů na dvě kategorie podle toho, zda se přejmenovávat mají či nikoliv, protože kdyby se v tomto případě anonymní uzly nepřejmenovávaly, výsledek by byl opět chybný, neboť všechny oddělení by sdílela stejný anonymní uzel.

2. *Vytváření po trojicích*: Jinou možností je vytvářet RDF data postupně po trojicích s pomocí konstruktoru trojice, do kterého se dosazuje hodnota subjektu, predikátu a objektu. Tento přístup využívá například jazyk XsRQL.

```
ex:department_01 ex:persons _:blank1.
_:blank1 ex:in ex:person_01.

ex:department_01 ex:persons _:blank2.
_:blank2 ex:in ex:person_02.

ex:department_02 ex:persons _:blank3.
_:blank3 ex:in ex:person_03.
```

Příklad 2.32: „Chybný“ výsledek vytváření množiny zaměstnanců

```
union(traverse((ex:department - ex:employees -> true),
        rdf:rest, vtrav:forward, vtrav:transitive),
        (ex:department - ex:employees -> true))
        - rdf:first -> true
```

Příklad 2.33: Výběr prvků seznamu v jazyce Versa

3. *RDF graf jako důkaz*: Jinou možností je přístup, kdy je výsledkem dotazu kromě samotného výsledku také RDF graf představující „důkaz“. Vrácený RDF graf v tomto případě představuje takovou podmnožinu vstupních dat, pro kterou daný dotaz vrací stejný výsledek.

4. *Popis výsledku*: Jiným způsobem implicitního vytvoření grafu jsou DESCRIBE dotazy jazyka SPARQL, kdy jsou krom požadovaných dat vrácena navíc ještě data nějak popisující vrácený výsledek.

   Tento typ dotazů nebyl formálně popsán. Možný přístup je ten, že pokud výsledek obsahuje jako objekt anonymní uzel, zahrnou se do výsledku i trojice obsahující tento anonymní uzel jako subjekt. Přičemž tento postup se opakuje, dokud lze nějakou trojici přidat[19].

2.10 Výběr dat s rekurzivně definovanou strukturou

Zatím probrané způsoby výběru dat se zabývaly pouze výběrem jednoduše strukturovaných dat. Struktura některých dat však může být například definována rekurzivně. Příkladem může být seznam z rdf: slovníku. Uzel seznamu (typu `rdf:List`) obsahuje predikát `rdf:first` určující prvek seznamu a predikát `rdf:rest` (s oborem hodnot `rdf:List`) určující zbytek seznamu. Prázdný seznam je označen uzlem `rdf:nil` (také typu `rdf:List`). Základní možnosti výběru, tak jak byly popsány v podkapitole 2.6, nedostačují na výběr prvků ze seznamu, proto jazyky často obsahují další konstrukce, které základní výběr dat zesilují. Příkladem mohou být:

1. *Tranzitivní relace*: Nejjednodušším zesílením základního výběru dat je schopnost jazyka označit některé predikáty za tranzitivní a nebo konstrukce umožňující přes nějaké predikáty tranzitivně přejít. Příkladem takové konstrukce může být funkce `traverse` jazyka Versa. Příklad 2.33 ukazuje výběr zaměstnanců ze seznamu, který je k oddělení `ex:department` připojen predikátem `ex:employees`.

2. *Regulární cesty*: Silnější konstrukcí je umožnit popis cest pomocí regulárních výrazů. Je dobré si uvědomit, že v takovýchto cestách nemá dobrý smysl použití proměnných. Respektive některé proměnné by pak musely reprezentovat množiny hodnot, eventuálně množiny množin hodnot a podobně. Použití regulárních výrazů pro popis cest se tedy zdá přímočaré hlavně u jazyků s navigačním přístupem k výběru dat. Přesto jazyk Versa ani jazyk XsRQL tuto možnost neobsahuje.

```
ex:department ex:employees/rdf:rest*/rdf:first ?member.
```

Příklad 2.34: Výběr prvků seznamu v jazyce ARQ

Pokud graf obsahuje	**přidej do grafu**
`uuu rdf:first xxx`	`uuu ex:content xxx`
`uuu rdf:rest vvv` `vvv ex:content xxx`	`uuu ex:content xxx`

Tabulka 2.2: Odvozovací pravidla predikátu `ex:content`

Jazyk Xcerpt umožňuje při popisu uzlu stromu (ve vzoru stromu) definovat, že tento uzel musí obsahovat podstrom, který je s daným uzlem spojen určitou cestou, která může být popsána pomocí regulárního výrazu.

Dalším jazykem využívajícím regulární cesty je jazyk ARQ[1]. Ten u vzoru trojice umožňuje místo predikátu uvést regulární výraz popisující cestu mezi subjektem a objektem. Popis se týká pouze predikátů na této cestě.

Vzor zajišťující výběr dat ze seznamu pomocí jazyka ARQ je ukázán v příkladu 2.34.

3. *Definice odvozování*: Jiné řešení je definovat odvozování. Například definovat predikát `ex:content` a odvozovací pravidla podle tabulky 2.2. Pak je možné vybírat prvky seznamu i za použití základního výběru. Tento způsob výběru dat podporují jazyky TRIPLE a Xcerpt.

   Možnost podpory odvozovacích pravidlech v jazycích je probírána v podkapitole 2.11.

4. *Obecná rekurze*: Umožňuje-li jazyk nějakou formu obecné rekurze, je možné ji samozřejmě využít pro průchod rekurzivní strukturou grafu.

Jazyky jako SPARQL, či SeRQL nenabízí žádné silnější prostředky kromě možnosti kombinovat vzory, resp. dotazy. Stejně tak jazyk XsRQL nenabízí žádné silnější prostředky.

Nejsilnějším prostředkem je (ve spojení s dalšími prostředky) obecná rekurze. Použití obecné rekurze však s sebou přináší úskalí v možnosti zapsat dotaz, který není možné v konečném čase vyhodnotit.

Možnost definovat predikát jako tranzitivní lze snadno popsat jak pomocí regulárních cest tak pomoci odvozování. Regulární cesty lze vyjádřit pomocí pravidel pro odvozování. Po obecné rekurzi je tedy podpora pro obecné odvozování druhou nejsilnější technikou. Pro praktické použití se však pro výběr dat zdají být regulární cesty pro svou jednoduchost dobrou volbou.

2.11 Podpora odvozování

Podobně jako k podpoře slovníků přistupují jazyky také k podpoře odvozování. Možnosti jsou:

1. *Žádná podpora*: Nejjednodušší jazyky nepodporují žádné odvozování. Některé přitom předpokládají, že podpora pro odvozování by se měla řešit na úrovni zdroje dat. Do této kategorie patří například jazyk SPARQL.

2. *Vestavěná podpora*: Mezi jazyky s vestavěnou podporou patří například jazyk SeRQL podporující rdfs: odvozování.

   Mnoho jazyků je vyvíjeno pro konkrétní úložiště a postrádají formální specifikaci. Je proto občas nesnadné určit, kdy je použití daného odvozování součástí jazyka a kdy jde pouze o vlastnost použitého úložiště.

3. *Obecná podpora*: Odvozování bývá definováno tak, že pokud graf G je pravdivý v každé interpretaci, ve které je pravdivý graf H, pak graf G lze odvodit (*entail*) z grafu H. Konkrétní odvozování je pak dáno podmínkami, které interpretace musí splňovat.

   Podpora odvozování na základě této definice je však velmi nepohodlná. Naštěstí většinu odvozování lze snadno popsat pomocí odvozovacích pravidel, která mají obecně podobu „pokud graf obsahuje tyto trojice, přidej do něj tyto trojice“. Graf G pak lze odvodit z grafu H, pokud konečnou aplikací odvozovacích pravidel na graf H lze obdržet graf, který je nadgrafem grafu G. Obecná podpora pro odvozovací pravidla se pak vyjadřuje mnohem snadněji.

   Obecné odvozování pěkně podporuje například jazyk TRIPLE. Mezi další jazyky s podporou pro obecné odvozování patří jazyk Xcerpt. V příkladu 2.35 je ukázka podpory rdfs: odvozování v jazyce TRIPLE. Pokud máme vstupní model pojmenovaný například `data` a chceme nad ním provést dotaz s použitím rdfs: odvozování, provede se to jednoduše tak, že se vzor doplní o specifikaci modelu `@rdfschema(data)`. Dotaz v jazyce TRIPLE, který ze vstupních dat vybere všechny trojice, včetně těch co z něj lze odvodit s použitím rdfs: odvozování, ukazuje příklad 2.36.

   Podpora odvozování je velmi užitečná vlastnost, neboť umožňuje s trojicemi popsanými pomocí odvozování pracovat zcela transparentně.

   Je dobré upozornit na jednu věc: uzavřené jazyky nejsou automaticky stejně silné jako jazyky s podporou pro obecné odvozování, neboť nejde jen o to přidat do grafu nové trojice, pokud obsahuje určité trojice (což lze v uzavřeném jazyce zařídit poddotazem), ale je třeba tento postup opakovat, dokud je možné nějaké trojice přidat.

```
rdf := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
rdfs := 'http://www.w3.org/2000/01/rdf-schema#'.
type := rdf:type.
subPropertyOf := rdfs:subPropertyOf.
subClassOf := rdfs:subClassOf.

FORALL Mdl @rdfschema(Mdl)
{
    transitive(subPropertyOf).
    transitive(subClassOf).

    FORALL O,P,V O[P->V] <-
            O[P->V]@Mdl.

    FORALL O,P,V O[P->V] <-
            EXIST S S[subPropertyOf->P] AND O[S->V].

    FORALL O,P,V O[P->V] <-
            transitive(P) AND EXIST W (O[P->W] AND W[P->V]).

    FORALL O,T O[type->T] <-
            EXIST S (S[subClassOf->T] AND O[type->S]).
}
```

Příklad 2.35: Popis odvozování rdfs: slovníku v jazyce TRIPLE

```
FORALL X,Y,Z <- X[Y->Z]@rdfschema(data).
```

Příklad 2.36: Použití odvozování v jazyce TRIPLE

2.12 Podpora datatypů

RDF nedefinuje vyjma datatypu `rdf:XMLLiteral` žádné standardní datatypy. S tímto nedostatkem se musí vypořádat jazyky, které umožňují použití aritmetiky. Možné přístupy jsou obecně dva:

1. *Slabě typované jazyky*: Tyto jazyky nehledí na datatyp literálu a jeho lexikální hodnotu se vždy snaží interpretovat podle kontextu, například jako číslo.

   Tento přístup vede k nutnosti zavést různé symboly či jména pro operátory běžně označované stejným symbolem, pracujícími však s různými typy dat. Příkladem může být operátor číselného a lexikografického porovnání či sčítání a konkatenace.

   Tento přístup je vhodný pouze pro ryze dotazovací jazyky, kde je aritmetika použita pouze v podmínkách a není tedy nutné zabývat se datatypem výsledku aritmetických operací. A nebo u jazyků neumožňujících vytváření nových trojic, kde lze absenci datatypu ve výsledku tolerovat.

   Slabě typovaným jazykem je například jazyk Versa.

2. *Silně typované jazyky*: Tyto jazyky provádí aritmetické operace na základě datatypu. Datatyp výsledku je tedy dán datatypem operandů. Většinou podporují XML Schema datatypy, jejichž použití je doporučováno přímo specifikací RDF.

   Protože však psaní literálů včetně datatypů je mnohdy zbytečně zdlouhavé, obsahují jazyky často podporu pro zkrácený zápis. Například v jazyce SPARQL (či ARQ) je možné číselné a logické literály zapisovat bez uvozovek a specifikace datatypu, přičemž odpovídající datatyp se doplní automaticky. Podobný přístup má také jazyk SeRQL. Ten se však navíc v případě, že pouze jeden z operandů má určený datatyp, pokouší přiřadit operandu bez datatypu stejný datatyp jako má operand s datatypem.

   Mezi silně typované jazyky, kromě již zmíněných, patří dále například jazyk XsRQL.

2.13 Možnosti jazyků obecně

Předchozí podkapitoly ukázaly, že dobrým (a také často používaným) přístupem k výběru dat je použití vzorů. Vzory mají mnoho výhod:

- Povaha vzorů je blízká vlastní povaze dat.
- Pomocí vzorů lze dobře popsat jak základní výběr tak i vytváření dat.
- Velmi snadno se vyjadřují složitější hodnotové podmínky a podmínky na nepovinnost dat, které se při navigačním přístupu vyjadřují dosti komplikovaně.
- Vzory popisují více podmínky řešení, než způsob jeho výpočtu.

Pokud se navíc jazyk doplní o takové prostředky, jakými jsou například regulární cesty (jazyk ARQ) a nebo podpora pro vyjádření obecného odvozování (jazyky TRIPLE a Xcerpt), dosahují jazyky při výběru dat dobrých výsledků.

Horších výsledků však jazyky obecně dosahují při vytváření nových dat. Mnoho jazyků nedovede nové trojice vytvářet vůbec. U těch, které to dokáží, lze spatřovat značnou nevyrovnanost mezi schopností data vybírat a data vytvářet. Například několik zde zmíněných jazyků (ARQ, Versa, TRIPLE a Xcerpt) je schopno získat prvky uložené v seznamu, ale už žádný z nich není schopen na základě těchto prvků seznam opět vytvořit.

Dotazovací jazyky pro RDF také většinou nebývají uzavřené. Z porovnávaných jazyků lze za uzavřený považovat jen jazyk Xcerpt, kde dotaz může jako vstup použít výstupy jiných dotazů nacházejících se v programu[2] ve formě pravidel, která odpovídají dotazům.

[2]Program se v jazyce Xcerpt skládá z dotazů (*goals*) a pravidel (*construct-query rules*).

Kapitola 3

Návrh jazyka

Tato kapitola se zabývá vytvořením nového dotazovacího jazyka pro datové úložiště Trisolda. Stručně představuje myšlenky, které stály za návrhem tohoto jazyka. Následně pak podrobně představuje syntaxi a sémantiku navrženého jazyka.

3.1 Volba přístupu

Základním požadavkem bylo vytvořit silný, uzavřený dotazovací jazyk, který by umožňoval nejen vybírat z různě strukturovaných dat, ale zároveň také různě strukturovaná data vytvářet.

Všechny zatím zmíněné prostředky, jakými byly například regulární cesty nebo podpora pro obecné odvozování, které dávají jazyku sílu při výběru dat, nelze dobře uplatnit při vytváření strukturovaných dat, například při vytváření seznamu (instance třídy `rdf:List`) z předem neznámé množiny uzlů. Bylo tedy třeba přijít se zcela jiným přístupem. Nově navrhované přístupy přitom vždy počítaly s využitím myšlenky vzorů spíše než s navigačním přístupem.

Zpočátku byly snahy o vytvoření silného jazyka spjaty s myšlenkou vytvořit konstrukci podobnou while cyklu, ovšem uzpůsobenou pro práci s grafy, a zároveň zachovat funkcionální povahu dotazovacího jazyka. Tato konstrukce měla opakovat tělo cyklu, dokud nalézala ve vstupním grafu daný podgraf (určený vzorem). Každé vykonání těla cyklu mělo mít možnost přidat nějaká data do vstupního grafu (bez této možnosti by konstrukce měla pouze sílu for cyklu[1]) a také určit data, která se mají vložit do celkového výsledku cyklu.

Třebaže je tento nápad v mnohém zajímavý, možnosti práce s takovouto konstrukcí nejsou příliš pohodlné. Přirozeným výsledkem této konstrukce je totiž pouze graf (a nebo množina či posloupnost grafů), což omezuje možnosti dalších konstrukcí při práci s takovýmto výsledkem.

To vedlo k myšlence, že prostředky jazyka nesmí být jen silné, ale že také do svých výsledků musí zavádět i jistou formu pořádku (hrubší členění výsledného grafu). Tato

[1] tedy pouze *primitivní rekurze*

myšlenka měla být naplněna úpravou jazyka Haskell[3]. Tento jazyk umožňuje definovat rekurzivní datové struktury. Tyto definice by se pak použily (jako vzor) k výběru dat a nebo jako konstruktor k tvorbě nových dat. Jednotlivé konstrukce jazyka by tak nevracely nějaký obecný graf, ale graf nějakého typu, tedy s nějakou definovanou strukturou.

Použití upraveného jazyka Haskell by vypadalo následovně. Definují se datové typy popisující strukturu grafů. V samotném programu se pak tyto datové typy použijí jako vzory pro získání dat z databáze, čímž se získají instance těchto typů. Tyto instance jsou buď přímo výsledkem a nebo se s nimi dále pracuje již pomocí standardních konstrukcí jazyka Haskell. Tento postup se však až příliš zabýval tím, jak řešení získat, než jak by mělo vypadat. Jinak řečeno, byl to více jazyk programovací než jazyk dotazovací.

Nejčastější způsob použití předpokládal, že výsledkem budou buď přímo získané instance nějakého datového typu a nebo nějaké jejich transformace získané prostým průchodem těmito instancemi, což vedlo k myšlence, že popis datového typu by měl zároveň určovat co udělat s daty po jejich nalezení.

To nakonec vedlo k návrhu jazyka založeného čistě na vzorech, kde vzory určují jak výběr tak vytváření dat, a který umožňuje vzory pojmenovávat, parametrizovat a odkazovat se na ně z jiných vzorů. Vzor se může rekurzivně odkázat sám na sebe, což jazyku dává sílu obecné rekurze. S možností obecné rekurze ovšem přichází nevýhoda, kterou je možnost zapsat dotaz, který nebude možné v konečném čase vyhodnotit. Navrhovaný dotazovací jazyk dostal později jméno Tequila (**T**risolda **Qu**ery **La**nguage).

Následující podkapitoly popisují syntax a sémantiku jazyka Tequila. Při popisu se snaží popisovat jednotlivé konstrukce jazyka v tom pořadí, v jakém byly do jazyka přidávány, včetně motivace pro jejich přidání. Zároveň jsou ale uspořádány tak, aby bylo možné jazyk Tequila snadno pochopit.

3.2 Základ jazyka

Za základ vzorů jazyka Tequila byly zvoleny vzory jazyka SPARQL, neboť působí jednoduchým dojmem a pomocí operátorů (jakými jsou například `union` či `optional`) je lze přirozeným způsobem skládat do složitějších vzorů. Stejně tak formát proměnných a komentářů, které začínají znakem # a pokračují až do konce řádku, je převzat z jazyka SPARQL. Na rozdíl od jazyka SPARQL záleží u klíčových slov jazyka Tequila na velikosti písmen.

V prologu každého dotazu jazyka Tequila se může volitelně vyskytovat definice prefixů. Ty se definují pomocí klíčového slova `prefix` stejným způsobem jako v případě jazyka SPARQL. Poté následuje samotný dotaz, který je uvozen klíčovým slovem `get`, za kterým následuje vzor grafu tvořící vlastní dotaz.

Jednoduchý dotaz, který ze vstupních dat vybere všechny instance třídy `rdfs:Class`, ukazuje příklad 3.1.

Třebaže se v základu vzory grafů jazyka Tequila příliš neliší od vzorů grafů jazyka SPARQL, z důvodu požadované uzavřenosti nemůže výsledkem dotazu být množina ohodnocení proměnných, jako je tomu v případě jazyka SPARQL. Za výsledek dotazu jazyka

```
prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

get
{
    ?res rdf:type rdfs:Class.
}
```

Příklad 3.1: Jednoduchý dotaz v jazyce Tequila

Tequila se proto bude považovat množina grafů, které, neformálně řečeno, ve vstupních datech odpovídají vzoru grafu použitému v dotazu. Takovou množinu grafů lze kdykoliv ztotožnit s grafem. Přesnější sémantika vyhodnocování dotazů bude představena v podkapitole 3.4.

3.3 Pojmenované vzory a výběr dat

Nejdůležitější vlastností jazyka Tequila, jak již bylo zmíněno v úvodu kapitoly, je možnost vzory pojmenovat a odkazovat se na ně. Pojmenované vzory se definují po prologu dotazu před samotným dotazem. Definice se skládá ze jména vzoru a seznamu formálních parametrů následovaných vzorem grafu, kterému je tímto způsobem přiřazeno jméno. Jako jméno vzoru se používá URI reference. Na pojmenovaný vzor se odkazuje pomocí klíčového slova `use` následovaného jménem vzoru a seznamem parametrů.

Dotaz, který vybere celý seznam zaměstnanců, jehož první uzel je hodnotou vlastnosti `ex:employees` uzlu `ex:department`, může vypadat například tak, jak je ukázáno na příkladu 3.2.

Fungování tohoto dotazu je intuitivně poměrně jasné. První větev vzoru union vybírá jeden článek seznamu určený parametrem `?N` a rekurzivně použije vzor `ex:list` na zbytek seznamu. Pokud je už zbytek seznamu prázdný, uspěje druhá větev vzoru union, která testuje, zda je parametr `?N` roven `rdf:nil`.

Při bližším pohledu na dotaz je však zřejmé, že sémantiku vyhodnocování dotazů jazyka SPARQL nebude možné použít. Přístup jazyka SPARQL k vyhodnocování dotazu by znamenal vyhodnotit samostatně vzor trojice `ex:department ex:employees ?list.` a dále pak pojmenovaný vzor `use ex:list(?list)` a provést (přirozené) spojení těchto vyhodnocení. Nemožnost použití tohoto způsobu vyhodnocování spočívá v tom, že při vyhodnocování pojmenovaného vzoru je mu třeba předat hodnotu parametru, tedy pojmenovaný vzor nelze vyhodnocovat nezávisle od ostatních (jemu předcházejících) vzorů. Proto bude třeba popsat vyhodnocování dotazů jiným způsobem.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ex:  <http://www.example.org/term#>

ex:list(?N)
{
    {
        ?N rdf:first ?F.
        ?N rdf:rest  ?R.
        use ex:list(?R)
    }
    union
    {
        filter ?N = rdf:nil.
    }
}

get
{
    ex:department ex:employees ?list.
    use ex:list(?list)
}
```

Příklad 3.2: Výběr seznamu v jazyce Tequila

3.4 Vyhodnocování dotazů

Přístup k vyhodnocování dotazu použitý jazykem Tequila je podobný přístupu jazyka RQL[6] či Prologu, který je založen na myšlence postupného hledání řešení s návratem.

Jak již bylo řečeno, dotaz se skládá z klíčového slova `get` a vzoru grafu. Výsledkem dotazu je posloupnost všech řešení tohoto vzoru grafu.

Vzor grafu se skládá z posloupnosti vzorů uzavřené ve složených závorkách. Hledání řešení vzoru grafu se provádí postupným hledáním řešení jeho podvzorů. Při nalezení řešení posledního podvzoru se vzor grafu považuje za vyřešený a jeho řešením je sjednocení[2] řešení jeho podvzorů.

Při nalezení řešení podvzoru může dojít k vazbě proměnných na určité hodnoty. Při dalším vyhodnocování (dokud není tato vazba zrušena) se pak s touto proměnnou pracuje jako s konstantou.

Pokud není možné nalézt řešení nějakého podvzoru, vrací se vyhodnocování na předchozí podvzor a hledá se jiné jeho řešení. Při tom se ruší vazby proměnných provedené při nalezení zamítnutého řešení.

Při hledání dalšího řešení vzoru grafu se hledá další řešení jeho posledního podvzoru. V případě že neexistuje, postupuje se stejně jako při hledání prvního řešení vzoru grafu, tedy návratem na předchozí jeho podvzor.

Pokud první podvzor vzoru grafu už nemá další řešení, pak už neexistuje ani další řešení vzoru grafu.

Vzor grafu může obsahovat tyto druhy vzorů[3]:

- *Vzor trojice*: Při hledání řešení vzoru trojice se postupně prochází vstupní graf (reprezentovaný posloupností trojic) a hledá se taková trojice, která odpovídá danému vzoru. Po nalezení odpovídající trojice se tato trojice považuje za řešení vzoru trojice. Dále dosud nesvázané proměnné nalézající se ve vzoru trojice se svážou s hodnotami podle nalezené trojice.

  Při návratu a hledání dalšího řešení se pokračuje v prohledávání vstupního grafu od v pořadí další trojice. Pokud již žádná trojice ze vstupního grafu danému vzoru trojice neodpovídá, další řešení neexistuje.

- *Filtr*: Vzor filter je podmínka (výraz) uvozená klíčovým slovem `filter` a zakončená (podobně jako trojice) tečkou. Při hledání řešení se tato podmínka vyhodnotí a pokud je nepravdivá, vzor filter nemá řešení. Pokud je pravdivá, řešením vzoru filter je prázdný graf a další řešení již vzor filter nemá.

- *Vzor grafu*: Hledání řešení vzoru grafu se provádí postupným hledáním řešení jeho podvzorů, tak jak bylo již popsáno.

[2] Používá se obyčejné množinové sjednocení množin trojic, k přejmenovávání anonymních uzlů tedy nedochází.

[3] Další druhy vzorů budou představeny v následujících podkapitolách.

```
ex:department ex:employees _:blank.
_:blank rdf:first "Josef Novak".
_:blank rdf:rest rdf:nil.
```

Příklad 3.3: Vstupní data příkladu 3.2

- *Vzor union*: Vzor union se skládá ze dvou vzorů spojených klíčovým slovem `union` a jeho účelem je provádět sloučení řešení jeho dvou podvzorů. Řešení se hledá buď v prvním podvzoru (první větvi vzoru union) nebo v druhém podvzoru (druhé větvi vzoru union). Pokud již ani jeden ze vzorů nemá řešení, pak už ani vzor union nemá řešení.

- *Vzor optional*: Vzor optional je vzor uvozený klíčovým slovem `optional`. Pokud podvzor nemá řešení, je řešením vzoru optional prázdný graf a další řešení již neexistuje. Jinak hledání řešení vzoru optional odpovídá hledání řešení jeho podvzoru. Vzor optional má tedy vždy alespoň jedno řešení.

- *Pojmenovaný vzor*: Napřed se podle jména najde definice pojmenovaného vzoru. Pokud je vzor určen proměnnou, která není svázána s URI referencí, a nebo dané URI referenci neodpovídá žádná definice pojmenovaného vzoru, pak pojmenovaný vzor nemá žádné řešení.

  Před samotným hledáním řešení se proměnné pojmenovaného podvzoru uvedené v seznamu formálních parametrů svážou s hodnotami předávanými jako skutečné parametry. Hledání řešení pojmenovaného vzoru dále již odpovídá hledání řešení vzoru grafu, který byl daným jménem pojmenován. Proměnné vyskytující se v pojmenovaném vzoru jsou považovány za lokální. Vazba vnějších proměnných se tedy uvnitř pojmenovaného vzoru neprojeví, stejně jako vazba proměnných uvnitř pojmenovaného vzoru se neprojeví vně tohoto vzoru.

- *Vzor any*: Tento vzor již nemá ekvivalent v jazyce SPARQL. Vzor any má podobu vzoru uvozeného klíčovým slovem `any`. Hledání prvního řešení odpovídá hledání řešení jeho podvzoru. Hledání dalšího řešení skončí vždy neúspěchem. Vzor any má tedy vždy maximálně jeden výsledek.

- *Vnořený dotaz*: Řešením vnořeného dotazu je sloučení všech řešení tohoto dotazu. Další řešení tento podvzor nemá. Vnořený dotaz má tedy z pohledu vnějšího dotazu vždy právě jedno řešení.

Postup vyhodnocování dotazu si pro názornost ukážeme na našem dříve představeném příkladu 3.2, přičemž vyhodnocování dotazu se bude provádět pro vstupní data uvedená v příkladu 3.3. Vyhodnocení dotazu proběhne postupně v těchto krocích:

1. Vyhodnocování dotazu začíná hledáním prvního řešení vzoru grafu začínajícího na řádku 18.

2. Hledání řešení tohoto vzoru grafu začíná hledáním řešení vzoru trojice z řádku 19. Vzor trojice odpovídá hned první trojici ve vstupním grafu, která se tak stává jeho řešením. Proměnná `?list` se váže podle nalezené trojice s hodnotou `_:blank`.

3. Hledání řešení vzoru grafu pokračuje hledáním řešení pojmenovaného vzoru z řádku 20. Lokální proměnná `?N` pojmenovaného vzoru se váže s hodnotou proměnné `?list`, tedy s hodnotou `_:blank`. Hledání řešení pojmenovaného vzoru odpovídá hledání řešení vzoru grafu začínajícího na řádku 5 a sestává z těchto kroků:

   (a) Hledání řešení pojmenovaného vzoru začíná hledáním řešení vzoru union z řádku 11.

   (b) Řešení vzoru union se hledá napřed v jeho první větvi, kterou představuje vzor grafu začínající na řádku 6.

   (c) Hledání řešení tohoto vzoru grafu začíná hledáním řešení vzoru trojice z řádku 7, který má pro aktuální vazby proměnných podobu `_:blank rdf:first ?F.`. Vzoru odpovídá druhá vstupní trojice, která se tak stává jeho řešením. Proměnná `?F` se váže na hodnotu `"Josef Novak"`.

   (d) Pokračuje se hledáním řešení vzoru trojce z řádku 8, který má pro aktuální vazby proměnných podobu `_:blank rdf:rest ?R.`. Vzoru odpovídá třetí vstupní trojice, která se tak stává jeho řešením. Proměnná `?R` se váže na hodnotu `rdf:nil`.

   (e) Posledním podvzorem vzoru grafu tvořícího první větev vzoru union je pojmenovaný vzor z řádku 9. Lokální proměnná `?N` pojmenovaného vzoru (do kterého se vstupuje) se sváže s hodnotou proměnné `?R`, tedy s hodnotou `rdf:nil`. Hledání řešení pojmenovaného vzoru odpovídá hledání řešení vzoru grafu začínajícího na řádku 5. Jedná se o druhý, rekurzivní vstup do tohoto pojmenovaného vzoru.

      i. Hledání řešení pojmenovaného vzoru začíná hledáním řešení vzoru union z řádku 11.

      ii. Řešení vzoru union se hledá napřed v jeho první větvi, kterou představuje vzor grafu začínající na řádku 6.

      iii. Hledá se tedy řešení vzoru trojice z řádku 7, který má pro aktuální vazby proměnných podobu `rdf:nil rdf:first ?F.`. Takovému vzoru neodpovídá žádná vstupní trojice.

      iv. První podvzor vzoru grafu představující první větev vzoru union nemá tedy řešení, tedy řešení nemá ani tento vzor grafu.

      v. První větev vzoru union tedy nemá řešení, pokračuje se tedy hledáním řešení v druhé větvi vzoru union.

      vi. Druhou větev vzoru union tvoří vzor grafu začínajícím na řádku 12.

      vii. Hledá se tedy řešení vzoru filter z řádku 13. Podmínka má pro aktuální vazby proměnných podobu `rdf:nil = rdf:nil`, takže je pravdivá. Řešením je tedy prázdný graf.

viii. Řešením vzoru grafu je tedy prázdný graf.

ix. Řešením druhé větve vzoru union a tedy i celého vzoru union je tudíž prázdný graf.

x. Řešením pojmenovaného vzoru, které odpovídá řešení vzoru grafu tvořeného pouze vzorem union, je tedy pouze řešení vzoru union, tedy prázdný graf.

(f) Vyřešením pojmenovaného vzoru byl vyřešen poslední podvzor vzoru grafu, který začíná na řádku 6. Řešení tohoto vzoru grafu je rovno sjednocení řešení jeho podvzorů, tedy grafu:

```
_:blank rdf:first "Josef Novak".
_:blank rdf:rest rdf:nil.
```

(g) Řešení tohoto vzoru grafu odpovídá řešení první větve vzoru union a tedy i řešení celého vzoru union.

(h) Řešením pojmenovaného vzoru, které odpovídá řešení vzoru grafu začínajícího na řádku 5 tvořeného pouze vzorem union, je tedy řešení vzoru union, tedy výše zmíněný graf:

```
_:blank rdf:first "Josef Novak".
_:blank rdf:rest rdf:nil.
```

4. Vyřešením pojmenovaného vzoru byl vyřešen poslední podvzor vzoru grafu, který začíná na řádku 18. Řešení tohoto vzoru grafu je rovno sjednocení řešení jeho podvzorů, tedy grafu:

```
ex:department ex:employees _:blank.
_:blank rdf:first "Josef Novak".
_:blank rdf:rest rdf:nil.
```

Toto řešení představuje první řešení dotazu.

5. Pokračuje se hledáním dalšího řešení dotazu. Hledá se tedy další řešení pojmenovaného vzoru z řádku 20, coby posledního podvzoru vzoru grafu tvořícího dotaz.

(a) Hledá se další řešení vzoru union z řádku 11, což je jediný podvzor vzoru grafu odpovídající pojmenovanému vzoru.

(b) Další řešení vzoru union se hledá v jeho první větvi, která jako poslední vrátila řešení.

(c) Hledání tedy pokračuje hledáním dalšího řešení vzoru grafu začínajícího na řádku 6.

(d) Hledá se tedy další řešení pojmenovaného vzoru z řádku 9.

i. Hledá se další řešení vzoru union z řádku 11, což je jediný podvzor vzoru grafu odpovídající pojmenovanému vzoru.

ii. Další řešení vzoru union se hledá v jeho druhé větvi, která jako poslední vrátila řešení.
iii. Hledá se tedy další řešení vzoru grafu začínajícího na řádku 12.
iv. Což odpovídá hledání dalšího řešení vzoru filter z řádku 13. Vzor filter ale další řešení (nikdy) nemá.
v. Další řešení tedy nemá ani vzor grafu začínající na řádku 12, který tvoří druhou větev vzoru union.
vi. Vzor union tedy již další řešení nemá.
vii. Neexistuje tedy ani další řešení pojmenovaného vzoru.

(e) Pojmenovaný vzor tedy další řešení nemá a vyhodnocování se vrací k hledání dalšího řešení vzoru trojice z řádku 8. Zároveň se tímto návratem ruší vazba proměnné `?R`.

(f) Vzor trojice z řádku 8 žádné další řešení nemá a vyhodnocování se vrací k hledání dalšího řešení vzoru trojice z řádku 7. Zároveň se tímto návratem ruší vazba proměnné `?F`.

(g) Vzor trojice z řádku 7 žádné další řešení nemá.

(h) Žádné další řešení tedy nemá ani vzor grafu začínající na řádku 6.

(i) První větev vzoru union tedy již žádné další řešení nemá a pokračuje se hledáním řešení druhé větve vzoru union.

(j) Druhou větev vzoru union představuje vzor grafu začínající na řádku 12.

(k) Hledání řešení tohoto vzoru grafu začíná hledáním řešení vzoru filter z řádku 13.

(l) Filtr má pro aktuální vazby proměnných tvar `_:blank = rdf:nil`, vzor filter tedy nemá řešení.

(m) Řešení tedy nemá ani pojmenovaný vzor začínající na řádku 12, který představuje druhou větev vzoru union.

(n) Vzor union tedy nemá řešení.

(o) Neexistuje tedy ani další řešení pojmenovaného vzoru.

6. Protože pojmenovaný vzor nemá řešení, pokračuje výpočet hledáním dalšího řešení vzoru trojice z řádku 19.

7. Tento vzor trojice však již další řešení nemá.

8. Neexistuje tedy ani další řešení vzoru grafu začínajícího na řádku 18, neexistuje tedy ani další řešení dotazu.

```
get
{
    construct ?X ?X ?X.
}
```

Příklad 3.4: Vytvoření trojice

```
prefix ex: <http://www.example.org/term#>

get
{
    ?X ex:pred ?Y.

    construct ?Y ex:ipred ?X.
}
```

Příklad 3.5: Vytvoření inverzního predikátu

3.5 Vytváření nových dat

Zatím byly rozebírány pouze možnosti výběru dat. Data (trojice) je však třeba umět také vytvářet. Proto byl do jazyka Tequila zařazen vzor construct, který má podobu bloku uvozeného klíčovým slovem `construct` obsahujícího vzory trojic. Pokud blok obsahuje pouze jeden vzor trojice, je možné složené závorky tvořící blok vynechat. Vzor construct se v podstatě také, jako většina zatím představených vzorů, vyskytoval již v jazyce SPARQL, sloužil však pouze k závěrečnému převodu řešení z formy ohodnocení proměnných do formy RDF grafu. V jazyce Tequila se však jedná o plnohodnotný vzor, který lze použít kdekoliv, kde je možné použít vzor.

Za jediné řešení vzoru construct se považuje graf obsahující vzory trojic, ve kterých jsou proměnné nahrazeny hodnotami, s kterými jsou (aktuálně) svázány. Pokud vzor trojice obsahuje proměnnou, která není svázána s žádnou hodnotou, je tato proměnná svázána s unikátním anonymním uzlem. Například výsledkem dotazu v příkladu 3.4 je, bez ohledu na vstupní data, jedna trojice skládající se ze stejných anonymních uzlů.

Užitečnější je dotaz v příkladu 3.5, který vytváří predikát `ex:ipred` jako inverzní k predikátu `ex:pred`.

Třebaže tento dotaz opravdu vytvoří inverzí vlastnost `ex:ipred`, za zjevnou nevýhodu lze považovat fakt, že výsledek bude obsahovat také trojice s vlastností `ex:pred`. Obecně je v mnohých případech vhodné svázat pomocí vzoru proměnné s určitými hodnotami, ale samotné řešení tohoto vzoru do celkového výsledku nezapočítat.

Za tímto účelem byl do jazyka Tequila přidán vzor match, který má podobu vzoru uvozeného klíčovým slovem `match`. Existence řešení vzoru match je dána existencí řešení jeho podvzoru, hodnoutou řešení vzoru match (pokud existuje) je však vždy prázdný graf. Příklad 3.6 ukazuje použití vzoru match při definici predikátu `ex:ipred`.

```
prefix ex: <http://www.example.org/term#>

get
{
    match ?X ex:pred ?Y.

    construct ?Y ex:ipred ?X.
}
```

Příklad 3.6: Vylepšené vytvoření inverzního predikátu

```
prefix ex: <http://www.example.org/term#>

get
{
    construct ex:department ex:employees _:blank.

    use ex:createList(_:blank)
}
```

Příklad 3.7: Vytvoření strukturovaných dat – první možnost

3.6 Vytváření strukturovaných dat

Tato podkapitola se zabývá možnostmi vytvářet data se složitější strukturou. Uvažme, že chceme vytvořit seznam a ten určitou vlastností připojit k nějakému uzlu. Například vlastností `ex:employees` k uzlu `ex:department`. Předpokládejme, že tento seznam nám nějakým způsobem vytvoří pojmenovaný vzor `ex:createList`. Nyní však stojíme před problémem, jak tento seznam připojit k uzlu `ex:department`. Na první pohled se nabízí několik možností.

První možností je spojit uzel `ex:department` s anonymním uzlem a ten předat vzoru `ex:createList` jako parametr, který má představovat první uzel vytvářeného seznamu. Tuto možnost ukazuje příklad 3.7.

Slabým místem tohoto řešení je případ, kdy by `ex:createList` měl vrátit prázdný list, tedy případ, kdy by celkové řešení mělo mít následující podobu:

```
ex:department ex:employees rdf:nil.
```

Druhou možností, jak k problému přistoupit, je předat vzoru `ex:createList` jako parametry uzel `ex:department` a vlastnost `ex:employees` a nechat vše na pojmenovaném vzoru, jak ukazuje příklad 3.8. Tento přístup by již fungoval i pro prázdný seznam, ale zbytečně moc svazuje pojmenovaný vzor `ex:createList` s tím, jak ho v tomto případě chceme použít. Pokud bychom jako výsledek dotazu chtěli například vracet pouze seznam zaměstnanců, tak bychom vzor `ex:createList` v této podobě použít nemohli.

```
prefix ex: <http://www.example.org/term#>

get
{
    use ex:createList(ex:department, ex:employees)
}
```

Příklad 3.8: Vytvoření strukturovaných dat – druhá možnost

```
prefix ex: <http://www.example.org/term#>

get
{
    ex:createList(?list)

    construct ex:department ex:employees ?list.
}
```

Příklad 3.9: Vytvoření strukturovaných dat – správná možnost

Jako dobré řešení se jeví přístup, který by vzoru `ex:createList` umožnil kromě výsledného grafu vrátit také uzel, který představuje první uzel seznamu. Protože navrhovaný jazyk už tak má svou sémantikou blízko k jazyku Prolog, jako vhodné a přímočaré řešení se jeví použití principu vstupně/výstupních proměnných známý z tohoto jazyka. Dotaz by pak šel snadno zapsat tak, jak je ukázáno na příkladu 3.9. Pokud bychom chtěli jako výsledek vrátit pouze seznam zaměstnanců lze to snadno provést dotazem podle příkladu 3.10.

Umožnit použití výstupních proměnných si vyžádá drobné rozšíření sémantiky hledání řešení pojmenovaného vzoru. Před hledáním řešení pojmenovaného vzoru se proměnné pojmenovaného podvzoru uvedené v seznamu formálních parametrů svážou s hodnotami předávanými jako skutečné parametry. Pokud je však parametrem dosud nesvázaná proměnná, žádná vazba formálního parametru se neprovede. Poté se hledá řešení pojmenovaného vzoru dříve popsaným způsobem. Po nalezení řešení se ty nesvázané proměnné, které byly předávány jako parametr, svážou (pokud je to možné) s hodnotou, se kterou byl při

```
prefix ex: <http://www.example.org/term#>

get
{
    ex:createList(?list)
}
```

Příklad 3.10: Samotné použití vytvoření strukturovaných dat

```
prefix ex: <http://www.example.org/term#>

get
{
    {
        ex:a ex:b ?X.
    }
    from construct
    {
        ex:a ex:b ex:c.
        ex:e ex:f ex:g.
    }
}
```

Příklad 3.11: Použití vzoru from

nalezení řešení svázán odpovídající formální parametr. Pokud jsou požadavky na vazbu proměnných protichůdné (což se může stát v případě, kdy byla jedna nesvázaná proměnná použita v seznamu parametrů vícekrát), je nalezené řešení zamítnuto.

Vstupní a výstupní proměnné se tedy syntakticky nikterak nerozlišují a použití proměnné jako vstupní či výstupní je dáno pouze jejím stavem před započetím hledání řešení pojmenovaného vzoru.

3.7 Skládání dotazů

Jednou z požadovaných vlastností navrhovaného jazyka je umožnit použít výsledek jednoho dotazu jako vstupní graf pro jiný dotaz. V jazyce Tequila je toho docíleno ještě obecněji s použitím vzoru from, který má podobu dvou vzorů spojených klíčovým slovem `from`. Hledání řešení vzoru from probíhá tak, že se napřed najde řešení vzoru za klíčovým slovem `from` a to se použije jako vstupní graf pro vzor před klíčovým slovem `from`, který se řeší standardním způsobem a jeho řešení jsou řešeními celého vzoru from. Až když tento vzor nemá další řešení, hledá se další řešení vzoru za `from`, které se opět použije jako vstupní graf pro vzor před `from`. Pokud vzor za `from` již nemá další řešení, nemá již další řešení ani vzor from.

Použití je možné vidět na příkladu 3.11, který bez ohledu na vstupní data vrací výsledek `ex:a ex:b ex:c.`.

Aby se bylo možné (pomocí vzoru from) odkázat také na data jiného úložiště, obsahuje jazyk Tequila vzor source, který má podobu klíčového slova `source` následovaného URI referencí úložiště. Řešením je graf obsahující všechny trojice obsažené v daném úložišti. Další řešení vzor nemá. Místo konstantní URI reference lze použít také proměnnou. Pokud při hledání řešení není proměnná svázána s URI referencí a nebo URI reference neodkazuje na zdroj dat, nemá vzor source žádné řešení.

Typ(A)	Typ(B)	Typ(A+B)
číselný literál	číselný literál	číselný literál
literál	literál	jednoduchý literál
literál	URI reference	jednoduchý literál
URI reference	URI reference	URI reference
URI reference	literál	URI reference
anonymní uzel	RDF term	anonymní uzel

Tabulka 3.1: Typy operandů operace +

3.8 Výrazy

Výrazy jazyka Tequila vychází z výrazů jazyka SPARQL. Přejaty byly pouze základní operátory.

Nad rámec operátorů jazyka SPARQL definuje jazyk Tequila operátor konkatenace +. Tento operátor lze použít kromě (nečíselných) literálů také pro URI reference a anonymní uzly. Výsledný typ operátoru konkatenace je dán typem levého argumentu. Pravým argumentem může být anonymní uzel pouze v případě, kdy je i levý argument anonymním uzlem. Možné kombinace typů operandů operace + (ve smyslu sčítání i konkatenace) a výsledný typ ukazuje tabulka 3.8. Jiné kombinace vedou k typové chybě.

Je důležité podotknout, že jméno anonymního uzlu je vždy interní a nelze tedy nikdy činit žádné předpoklady o jméně anonymního uzlu, pochopitelně kromě jmen anonymních uzlů zapsaných přímo v dotazu. Více se prací s anonymními uzly zaobírá podkapitola 3.9.

Dalším přidaným operátorem je binární operátor `like`, který byl inspirován operátorem `LIKE` jazyka SeRQL. Jedná se o binární operátor, jehož druhým operandem je jednoduchý literál (*simple literal*)[4]. Operátor vrací hodnotu `xsd:true`, pokud první operand není anonymním uzlem a po převodu na textový literál odpovídá regulárnímu výrazu, který je určen hodnotou druhého operandu. Jinak vrací hodnotu `xsd:false`.

Jazyk Tequila umožňuje použít ve výrazech vnořený dotaz jako primární výraz. Pokud je výsledkem vnořeného dotazu (po sloučení všech jeho řešení) prázdný graf, je následkem výrazu hodnota `false^^xsd:boolean`, pokud je výsledkem graf obsahující pouze jednu trojici, je výsledkem výrazu objekt[5] této trojice. Pokud graf obsahuje více trojic, je výsledkem typová chyba[6] (*type error*).

Jazyk Tequila umožňuje použít výrazy všude tam, kde je možné použít konstantní literál, tedy například při předávání parametrů a nebo ve vzorech trojic. Při použití ve vzorech trojic je třeba ovšem výraz uzavřít do závorek. Pokud při vyhodnocování takového výrazu je hodnotou typová chyba, daný vzor nemá řešení.

[4]Pro jiné typy druhého operandu je výsledkem typová chyba.

[5]Důvod, proč se za výsledek považuje zrovna objekt trojice je ten, že ten jako jediný může být jak URI referencí tak anonymním uzlem či literálem.

[6]Ve smyslu specifikace výrazů jazyka SPARQL.

Je dobré upozornit, že vzor `?X ?Y ?Z.` není ekvivalentní vzoru `(?X) ?Y ?Z.`, neboť v druhém případě před samotným vyhodnocením vzoru trojice dojde napřed k vyhodnocení výrazu `(?X)`, což v případě, že proměnná `?X` není svázána s žádnou hodnotou, povede k chybě.

3.9 Práce s anonymními uzly

Tato podkapitola shrnuje, jak jazyk Tequila pojímá anonymní uzly a jak s nimi pracuje.

Jméno anonymního uzlu použité ve vzoru identifikuje vždy konkrétní uzel, nejedná se tedy o existenční kvalifikaci jako v jazyce SPARQL. Jazyk Tequila však chápe jména anonymních uzlů vždy jako lokální pro konkrétní zdroj dat (čímž se liší od přístupu jazyka SeRQL). Přesněji řečeno, s každým jménem anonymního uzlu je implicitně spjato také (jedinečné) jméno zdroje dat, ve kterém se anonymní uzel vyskytuje. Přitom všechny anonymní uzly zapsané přímo ve vzorech tvoří jeden samostatný zdroj dat. Anonymní uzly vytvářené vzorem construkt pak (z pohledu původu anonymních uzlů) tvoří další samostatný zdroj dat.

Při konkatenaci anonymních uzlů se konkatenují také informace o těchto zdrojích dat. Pokud se při této konkatenaci vedle sebe dostanou zdroje dat „dotaz“, splynou v jeden. To má za následek, že například anonymní uzly `_:blank0` a `(_:blank + 0)`, pokud jsou zapsány přímo v dotazu, jsou totožné. Pokud ale například nějaká databáze obsahuje anonymní uzly `_:blank` a `_:blank0`, pak z anonymního uzlu `_:blank` získaného z této databáze není možné pomocí konkatenace nikdy získat anonymní uzel `_:blank0` z této databáze.

To dohromady vede k jednoduchému pravidlu. Uživatel nikdy nemůže činit žádné předpoklady o jménech anonymních uzlů, kromě těch jmen, které jsou přímo zapsány ve vzorech, čímž zůstala zachována hlavní myšlenka anonymních uzlů.

Při slučování grafů jazyk Tequila anonymní uzly nepřejmenovává (díky implicitní informaci o původu anonymního uzlu to není třeba).

Výše popsaný přístup jazyka Tequila k anonymním uzlům umožňuje úspěšně vyřešit všechny problémy popsané v podkapitole 2.8.

To, že nějaký anonymní uzel se má při vytváření dat vytvořit pro každého zaměstnance (tedy pro každé řešení vzoru construct) a nebo pro každé oddělení (tedy pro ta řešení vzoru construct, která se týkají stejného oddělení), se vyjádří jednoduše tak, že základní jméno anonymního uzlu se konkatenuje s identifikací toho, ke komu má anonymní uzel patřit, jak to ukazuje příklad 3.12 vytvářející jména osob a příklad 3.13 vytvářející množinu zaměstnanců oddělení.

```
prefix ex: <http://www.example.org/term#>

get
{
    match
    {
        # Výběr ?first_name a ?surname
    }

    construct
    {
        ?person ex:name (_:blank + ?person).
        (_:blank + ?person) ex:first   ?first_name.
        (_:blank + ?person) ex:surname ?surname.
    }
}
```

Příklad 3.12: Vytvoření jmen zaměstnanců

```
prefix ex: <http://www.example.org/term#>

get
{
    match
    {
        # Výběr ?department a ?person
    }

    construct
    {
        ?department ex:persons (_:blank + ?department).
        (_:blank + ?department) ex:in ?person.
    }
}
```

Příklad 3.13: Vytvoření množiny zaměstnanců

3.10 Moduly

Jazyk Tequila umožňuje zapsat definice často používaných pojmenovaných vzorů do speciálního souboru a tyto definice načíst pomocí direktivy `import` parametrizované jménem tohoto souboru.

Soubor může obsahovat definici prefixů i další direktivy `import`. Definice prefixů je vždy lokální pro daný soubor.

3.11 Syntaktický cukr

Tato podkapitola představuje několik konstrukcí a zkratek, které slouží k zpřehlednění či zjednodušení kódu zapsaného v jazyce Tequila.

3.11.1 Vzor where

Občas bývá přehlednější uvést podmínky na hodnoty proměnných až za použití těchto proměnných. Tento zápis umožňuje v jazyce Tequila vzor where, který spojuje dva vzory. Přičemž vzor

```
vzor_1
where
vzor_2
```

je ekvivalentní vzoru

```
{
    match vzor_2
    vzor_1
}
```

3.11.2 Samostatné vrcholy

Některé vzory vrací výsledek, který je ze své povahy spíše množinou uzlů, než množinou trojic. Proto jazyk Tequila obsahuje možnost zapsat kromě vzoru trojice také vzor uzlu. Využití může tato konstrukce nalézt například v pojmenovaném vzoru, který provádí konverzi kontejneru na množinu uzlů, jak ukazuje příklad 3.14. Způsob použití tohoto vzoru pak ukazuje příklad 3.15.

Aby bylo možné na graf stále pohlížet jako na množinu trojic a nebylo tedy nutné upravovat použitý datový model, je vzor uzlu `?node.` pouze zkráceným zápisem vzoru trojice `tql:shadowSubject tql:shadowPredicate ?node.`.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ex:  <http://www.example.org/term#>

ex:inContainer(?bag)
{
    get
    {
        construct
        {
            ?item.
        }
        where
        {
            ?bag ?pred ?item.
            filter ?pred like ("" + rdf: + "_[0-9]\+").
        }
    }
}
```

Příklad 3.14: Konverze kontejneru

```
prefix ex: <http://www.example.org/term#>

get
{
    ?x. from use ex:convert(ex:bag)
}
```

Příklad 3.15: Použití konverze kontejneru

3.11.3 Vynechání závorek

V některých případech umožňuje jazyk Tequila vynechat složené závorky. Prvním případem byl vzor construct, který umožňuje vynechat závorky tvořící blok se vzory trojic, pokud tento obsahuje pouze jeden vzor trojice.

Druhým případem je dotaz, kde mohou být vynechány složené závorky vzoru grafu, pokud má dotaz tuto podobu:

```
get
{
    vzor_grafu_1
    where
    vzor_grafu_2
    from
    vzor_grafu_3
}
```

Přičemž část where a nebo část from nemusí být přítomna. Tento zkrácený zápis umožňuje psát dotazy ve formě:

```
get
vzor_grafu_1
where
vzor_grafu_2
from
vzor_grafu_3
```

3.12 Celkový pohled na vzory

V tuto chvíli již byly představeny všechny vzory, které je možné v jazyce Tequila zapsat, a to včetně vzorů tvořících syntaktický cukr. Za základní lze považovat vzor trojice, vzor filter, vzor construct a použití pojmenovaného vzoru. Klíčová slova `get`, `from`, `match`, `optional`, `union`, `any` a `where` lze pak považovat ze operátory, které tyto základní vzory skládají do podoby složitějších vzorů. Pro snadnější použití je dobré určit prioritu těchto operátorů. Ta byla určena následovně:

1. `any`, `match`, `optional`, `get`
2. `where`
3. `from`
4. `union`

Přičemž menší číslo znamená vyšší prioritu.

3.13 Výsledné možnosti jazyka

Všechny konstrukce jazyka Tequila již byly popsány. Celkové možnosti jazyka budou nyní demonstrovány na již složitějším příkladu 3.16, který definuje pojmenovaný vzor převádějící kontejner, určený parametrem `?bag`, na seznam, který je vrácen jako výsledek a jehož první uzel je vrácen skrze parametr `?list`.

Převod probíhá následujícím způsobem. Pokud je kontejner prázdný (a jen když je prázdný), uspěje druhá větev vzoru union začínající na řádku 25 a sváže výstupní parametr `?list` s hodnotou `?rdf:nil` (řádek 28). Pokud kontejner není prázdný, uspěje první větev vzoru union začínající na řádku 6. Ta pomocí vzoru any (řádek 18) vybere jeden prvek z kontejneru. Poté rekurzivně volá pojmenovaný vzor na kontejner (řádek 8), z kterého byl odebrán vybraný prvek. Poté z vybraného prvku a ze seznamu vytvořeného rekurzivním voláním vytvoří nový seznam (řádky 15 a 16).

Mezi další schopnosti jazyka Tequila patří například možnost definovat si pojmenované vzory nahrazující absenci přímé podpory agregačních funkcí (viz dodatek A) a nebo absenci podpory slovníku, například rdfs: slovníku (viz dodatek B).

```
prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ex:   <http://www.example.org/term#>

ex:convert(?bag, ?list)
{
    {
        {
            use ex:convert(?bag, ?sublist)
            from get
            {
                ?bag ?Y ?Z.
                filter ?Y != ?pred && ?Z != ?item.
            }

            construct ?list rdf:first ?item.
            construct ?list rdf:rest  ?sublist.
        }
        where any
        {
            ?bag ?pred ?item.
            filter ?pred like ("" + rdf: + "_[0-9]+").
        }
    }
    union
    {
        filter !get{?bag ?pred ?item.}.

        match ?list.from construct {rdf:nil.}
    }
}
```

Příklad 3.16: Konverze kontejneru na seznam

Kapitola 4

Ukázkové dotazy

Tato kapitola se snaží ukázat možnosti použití jazyka Tequila na základě ukázek řešení praktických dotazů.

4.1 Výběr dotazů

Jako ukázkové dotazy byly zvoleny dotazy, které byly použity v práci *RDF Querying: Language Constructs and Evaluation Methods Compared*[19]. Ukázková vstupní data a dotazy byly pouze drobně upraveny za účelem dosažení větší přesnosti.

Dotazy se týkají jednoduchého knihovního systému, jehož schéma je uvedeno v příkladu 4.1, vlastní data pak v příkladu 4.2. K datům byly přidány informace o knize „The Civil War“, pro identifikaci knihy „Bellum Civile“ bylo zvoleno místo anonymního uzlu URI `ex:bellum_civile`. Datový typ vlastnosti `ex:year` byl změněn z typu `xsd:gYear` na typ `xsd:integer`, neboť cílem není demonstrovat šíři podpory datatypů, ale použití agregačních funkcí.

4.2 Dotazy a jejich řešení

Jako ukázka možností jazyka Tequila byly zvoleny následující dotazy.

4.2.1 Výběrové dotazy

Dotaz č. 1: *„Vyberte všechny eseje (instance třídy ex:Essays) společně se jmény jejich autorů.“*

Dotaz řeší příklad 4.3, přičemž využívá základní podporu pro RDF Schema (viz dodatek B). Pomocí vzoru trojice na vlastnost `rdf:type` (řádek 9) prochází všechny instance a pomocí pojmenovaného vzoru `rdfs:subClassesOf` (řádek 10) testuje, zda je instance typu `ex:Essay`. Pro esej pak pomocí vnořeného dotazu (řádek 12) nalezne všechny její autory. Použití vnořeného dotazu místo obyčejného vzoru grafu zajistí, že každá esej bude ve svém řešení obsahovat všechny své autory.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ex:   <http://www.example.org/term#>.

ex:Writing  rdfs:label  "Novel".
ex:Writing  rdf:type  rdfs:Class.
ex:translator  rdfs:range  foaf:Person.
ex:translator  rdfs:domain  ex:Writing.
ex:translator  rdf:type  rdfs:Property.
ex:Novel  rdfs:subClassOf  ex:Writing.
ex:Novel  rdfs:label  "Novel".
ex:Novel  rdf:type  rdfs:Class.
ex:Historical_Novel  rdfs:subClassOf  ex:Novel.
ex:Historical_Novel  rdfs:subClassOf  ex:Essay.
ex:Historical_Novel  rdfs:label  "Historical Novel".
ex:Historical_Novel  rdf:type  rdfs:Class.
ex:Historical_Essay  rdfs:subClassOf  ex:Essay.
ex:Historical_Essay  rdfs:label  "Historical Essay".
ex:Historical_Essay  rdf:type  rdfs:Class.
ex:Essay  rdfs:subClassOf  ex:Writing.
ex:Essay  rdfs:label  "Essay".
ex:Essay  rdf:type  rdfs:Class.
ex:author  rdfs:range  foaf:Person.
ex:author  rdfs:domain  ex:Writing.
ex:author  rdf:type  rdfs:Property.
```

Příklad 4.1: Schéma ukázkových dat

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix foaf: <http://xmlns.org/foaf/0.1/>.
@prefix ex:   <http://www.example.org/term#>.

ex:bellum_civile  rdf:type  ex:Historical_Essay.
ex:bellum_civile  ex:translator  _:b6.
ex:bellum_civile  ex:title  "Bellum Civile".
ex:bellum_civile  ex:author  _:b5.
ex:bellum_civile  ex:author  _:b4.
_:b1  rdf:type  ex:Historical_Novel.
_:b1  ex:year  "1990"^^xsd:integer.
_:b1  ex:title  "The First Man in Rome".
_:b1  ex:author  _:b3.
_:b2  rdf:type  ex:Historical_Essay.
_:b2  ex:title "The Civil War".
_:b2  ex:author  _:b4.
_:b3  foaf:name  "Colleen McCullough".
_:b4  foaf:name  "Julius Caesar".
_:b5  foaf:name  "Aulus Hirtius".
_:b6  foaf:name  "J. M. Carter".
```

Příklad 4.2: Ukázková data

```
1  prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  prefix foaf: <http://xmlns.org/foaf/0.1/>
4  prefix ex:   <http://www.example.org/term#>
5  import "schema.tql"
6
7  get
8  {
9      ?book rdf:type ?type.
10     match ?type. from use rdfs:subClassesOf(ex:Essay)
11
12     get
13     {
14         ?book ex:author ?author.
15         ?author foaf:name ?name.
16     }
17 }
```

Příklad 4.3: Řešení dotazu č. 1 v jazyce Tequila

```
1  prefix ex: <http://www.example.org/term#>
2
3  ex:closure(?subj)
4  {
5      get
6      {
7          ?subj ?pred ?obj.
8
9          use ex:closure(?obj) from
10         get
11         {
12             ?X ?Y ?Z.
13             filter ?X != ?subj.
14         }
15     }
16 }
17
18 get
19 {
20     {
21        use ex:closure(?book)
22     }
23     where
24     {
25         ?book ex:title "Bellum Civile".
26     }
27 }
```

Příklad 4.4: Řešení dotazu č. 2 v jazyce Tequila

4.2.2 Extrakční dotazy

Dotaz č. 2: *„Vyberte všechny uzly, které jsou v nějakém vztahu ke knize ‚Bellum Civile'.“* Úkolem tohoto dotazu je vybrat ze vstupního grafu podgraf obsahující ty uzly, které jsou dostupné z uzlu reprezentujícího knihu „Bellum Civile“.

Dotaz řeší příklad 4.4, a to pomocí pojmenovaného vzoru `ex:closure` (řádek 3). Tento pojmenovaný vzor vrací podgraf dostupný z uzlu `?subj`, který je vzoru předán jako parametr. Pojmenovaný vzor vybírá uzly dostupné přímo z uzlu `?subj` (řádek 7) a na tyto uzly pak rekurzivně volá `ex:closure` (řádek 9). Aby se předešlo zacyklení, je vnořený dotaz rekurzivně volán nad vstupním grafem neobsahujícím již ve trojicích uzel `?subj` na pozici subjektu (řádek 10).

Pojmenovaný vzor vrací vždy právě jedno řešení, čehož je docíleno použitím vnořeného dotazu (řádek 5).

```
1  prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2  prefix ex:   <http://www.example.org/term#>
3
4  get
5  {
6      get
7      {
8          ?subj ?pred ?obj.
9
10         filter !(?subj like ("" + rdfs: + ".*")).
11         filter !(?pred like ("" + rdfs: + ".*")).
12         filter !(?obj  like ("" + rdfs: + ".*")).
13
14         filter ?pred != ex:translator.
15     }
16 }
```

Příklad 4.5: Řešení dotazu č. 3 v jazyce Tequila

```
1  get
2  {
3      ?subj ?pred ?obj. from dotaz1
4      filter !get { construct true
5                    where any ?subj ?pred ?obj.
6                    from dotaz2 }.
7  }
```

Příklad 4.6: Vyjádření rozdílu dotazů v jazyce Tequila

4.2.3 Redukční dotazy

Dotaz č. 3: *„Vyberte všechna data vyjma ontologie a informací o překladatelích.“*

Tento dotaz lze vyřešit snadno pomocí sady filtrů, jak ukazuje příklad 4.5. Podobné dotazy se někdy řeší jako rozdíl dvou dotazů. Takovou konstrukci jazyk Tequila přímo nepodporuje, je však schopen vyjádřit rozdíl dotazů pomocí filtru a vnořeného dotazu, jak ukazuje obecný příklad 4.6. V tomto dotazu se postupně iteruje přes trojice prvního dotazu (řádek 3) a pro každou takovou trojici se pomocí vzoru filter a vnořeného dotazu (řádek 4) testuje, zda není obsažena v druhém dotazu.

4.2.4 Restrukturalizační dotazy

Dotaz č. 4: *„Vytvořte vlastnost* `ex:authored` *jako inverzní k vlastnosti* `ex:author`*.“*

Řešení tohoto dotazu, jak ukazuje příklad 4.7, je velmi snadné. Aby byly všechny trojice popisující vlastnost `ex:authored` vráceny v jednom řešení, byl použit vnořený dotaz.

```
1  prefix ex: <http://www.example.org/term#>
2
3  get
4  {
5      get
6      {
7          construct ?author ex:authored ?book.
8          where ?book ex:author ?author.
9      }
10 }
```

Příklad 4.7: Řešení dotazu č. 4 v jazyce Tequila

```
1  prefix tql:  <http://ulita.ms.mff.cuni.cz/tequila/term#>
2  prefix foaf: <http://xmlns.org/foaf/0.1/>
3  prefix ex:   <http://www.example.org/term#>
4  import "aggregate.tql"
5
6  get
7  {
8      match ?author foaf:name "Colleen McCullough".
9
10     use tql:max() from  get
11     {
12         ?book ex:year ?year.
13         where
14         ?book ex:author ?author.
15     }
16 }
```

Příklad 4.8: Řešení dotazu č. 5 v jazyce Tequila

4.2.5 Agregační dotazy

Dotaz č. 5: *„Zjistěte poslední rok, ve kterém autorka ‚Colleen McCullough'*[1] *něco publikovala.“*

Dotaz řeší příklad 4.8 s využitím agregační funkce `tql:max` (viz dodatek A). Agregační funkce `tql:max` (řádek 10) je volána nad vstupním grafem tvořeným trojicemi s informacemi o roku publikování knih daného autora (řádek 10).

Dotaz č. 6: *„Pro každou podtřídu třídy ex:Writing spočítejte průměrný počet autorů na jednu knihu.“*

Toto je ukázka už o něco složitějšího dotazu. Možné řešení ukazuje příklad 4.9, využí-

[1] Jméno autora bylo vůči originálu změněno, neboť ukázková data neobsahovala u knih Julia Caesara informace o roku publikování.

vající jak základní podporu pro RDF Schema (viz dodatek B), tak agregační funkce (viz dodatek A).

Pomocí pojmenovaného vzoru `rdfs:subClassesOf` (řádek 10) se postupně iteruje přes všechny podtřídy třídy `ex:Writing` včetně této třídy (přičemž tuto možnost odstraňuje následující filtr). Pro každou takovou třídu se s použitím pojmenovaného vzoru `tql:avg` (řádek 15) spočítá průměrný počet autorů na jednu knihu. Tento výsledek je prostřednictvím vzoru match (řádek 13) přiřazen proměnné `?avg`. Ta je pak použita při vytvoření trojice s informací o průměrném počtu autorů na knihu v dané třídě (řádek 30). Důvodem, proč nebyl pojmenovaný vzor `tql:avg` rovnou použit jako výraz při vytváření této trojice, byla pouze snaha o větší přehlednost.

Průměrný počet autorů je počítán ze vstupních dat vytvořených poddotazem (řádek 15), který vybírá knihy dané třídy (řádek 17 a 18) a pomocí pojmenovaného vzoru `tql:count` (řádek 21) počítá pro každou knihu počet jejich autorů. Tento počet sváže s proměnnou `?count` (řádek 20) a přiřadí ho k dané knize vlastností `ex:count` (řádek 26), přičemž právě trojice s vlastností `ex:count` jsou výsledkem tohoto poddotazu.

To, že tento poddotaz sloužící jako vstup pro pojmenovaný vzor `tql:avg` vrací výsledek jako trojice s vlastností `tql:count` a ne jen samotné počty autorů, je velmi důležité, neboť vstupní graf je vždy množina, která kdyby obsahovala jen počty autorů, ztratila by se informace, kolikrát se tento počet autorů v řešení vyskytl, což by způsobilo chybu ve výpočtu průměru.

4.2.6 Dotazy na kombinování a odvozování

Dotaz č. 7: *„Zkombinujte informace o knize jménem ‚The Civil War‘od autora ‚Julius Caesar‘s knihou s identifikátorem ex:bellum_civile.“*

Příklad 4.10 ukazuje jedno z možných řešení, v němž vlastnosti jedné knihy jsou přiřazeny knize druhé a naopak. Tento způsob řešení předpokládá využití dotazu k doplnění vstupních dat pro jiný dotaz.

Dotaz č. 8: *„Vytvořte tranzitivní uzávěr relace `rdfs:subClassOf`.“*

Tento dotaz řeší snadno příklad 4.11 s využitím základní podpory pro RDF Schema (viz dodatek B). Díky vnořenému dotazu (řádek 7) je celý uzávěr vrácen jako jedno řešení dotazu.

Dotaz č. 9: *„Vytvořte vlastnost `co-author` dávající do vztahu autory, kteří spolupracovali na jedné knize.“*

Dotaz řeší příklad 4.12. Díky vnořenému dotazu (řádek 5) jsou všechny trojice popisující vlastnost `co-author` vráceny jako jedno řešení dotazu (jako jeden graf).

```
prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix tql:  <http://ulita.ms.mff.cuni.cz/tequila/term#>
prefix ex:   <http://www.example.org/term#>
import "schema.tql"
import "aggregate.tql"

get
{
    match ?class. from use rdfs:subClassesOf(ex:Writing)
    filter ?class != ex:Writing.

    match ?avg. from get
    {
        use tql:avg() from get
        {
            match ?book rdf:type ?type.
            match ?type. from use rdfs:subClassesOf(?class)

            match ?count. from
            use tql:count() from get
            {
                ?book ex:author ?author.
            }

            construct ?book ex:count ?count.
        }
    }

    construct ?class ex:avg ?avg.
}
```

Příklad 4.9: Řešení dotazu č. 6 v jazyce Tequila

```
prefix foaf: <http://xmlns.org/foaf/0.1/>
prefix ex:   <http://www.example.org/term#>

get
{
    get
    {
        match
        {
            ?book ex:title "The Civil War".
            ?book ex:author ?author.
            ?author foaf:name "Julius Caesar".
        }

        {
            construct ?book ?pred ?obj.
            where
            ex:bellum_civile ?pred ?obj.
        }
        union
        {
            construct ex:bellum_civile ?pred ?obj.
            where
            ?book ?pred ?obj.
        }
    }
}
```

Příklad 4.10: Řešení dotazu č. 7 v jazyce Tequila

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix ex:   <http://www.example.org/term#>
import "schema.tql"

get
{
    get
    {
        match ?class rdfs:subClassOf ?super.
        match ?subclass. from use rdfs:subClassesOf(?class)

        construct ?subclass rdfs:subClassOf ?super.
    }
}
```

Příklad 4.11: Řešení dotazu č. 8 v jazyce Tequila

```
prefix ex: <http://www.example.org/term#>

get
{
    get
    {
        construct
        {
            ?author1 ex:co-author ?author2.
        }
        where
        {
            ?book ex:author ?author1.
            ?book ex:author ?author2.
            filter ?author1 != ?author2.
        }
    }
}
```

Příklad 4.12: Řešení dotazu č. 9 v jazyce Tequila

Kapitola 5

Praktické srovnání jazyků

Cílem této kapitoly je porovnat jednotlivé jazyky na základě toho, jak dobře si vedou při řešení praktických dotazů. Za dotazy k porovnání byly zvoleny ukázkové dotazy z předchozí kapitoly doplněné o dotaz na vybrání všech prvků ze seznamu (dotaz č. 10) a o dotaz na vytvoření seznamu z prvků kontejneru (dotaz č. 11). Schopnosti jazyků vyjádřit jednotlivé dotazy jsou následující:

Dotaz č. 1: Pokud by se dotaz omezoval jen na přímé instance třídy `ex:Essay`, zvládly by ho dobře všechny zde porovnávané jazyky. Při nutnosti uvažovat i potomky třídy `ex:Essay` se jazyky SPARQL, SeRQL a XsRQL neobejdou bez vestavěné podpory pro rdfs: odvozování. Tuto podporu jazyk SPARQL standardně neobsahuje a jazyk XsRQL pravděpodobně[1] také ne.

Dotaz č. 2: Vybrat graf dostupný z jednoho uzlu zvládnou dobře jazyky ARQ, Xcerpt a Tequila, u nichž tento výběr má opravdu podobu grafu. Jazyk TRIPLE je schopen tento graf popsat pomocí modelu, vlastní výsledek ale vrací jen jako ohodnocení proměnných reprezentující subjekt, predikát a objekt jednotlivých trojic. Jazyk Versa dokáže pomocí funkce `traverse` vrátit pouze seznam uzlů dostupných z daného uzlu. Jazyky SPARQL, SeRQL a XsRQL jsou schopny provést výběr jen do omezené hloubky[2].

Dotaz č. 3: Tento dotaz dokáží vyjádřit všechny porovnávané jazyky, liší se však použitými prostředky. SeRQL, Versa, TRIPLE, Xcerpt a Tequila jsou schopny vyjádřit rozdíl dvou dotazů. Jazyky SPARQL, ARQ a XsRQL toho schopny nejsou a tento dotaz řeší pouze s pomocí podmínek na hodnotu uzlu. Jazyk Versa pracuje na úrovni seznamů, jeho výsledkem jsou v tomto případě seznamy subjektů, predikátů a objektů neobsahující ty ze schématu a neobsahující překladatele.

Dotazy č. 4 a 9: Jazyky SPARQL, SeRQL, ARQ a XsRQL dovedou tyto relace vytvořit v podobě RDF grafu, ten však již nemohou (jako definici relace) dále použít. Jazyky Tequila a Xcerpt jsou schopny tyto grafy vytvořit a také použít. Jazyky TRIPLE a Xcerpt jsou schopny popsat tyto relace pomocí odvozování a transparentně je použít v dotazech.

[1] Jeho popis se o použitém odvozování nezmiňuje.

[2] Tato hloubka je dána délkou cest použitých v konkrétních dotazech.

Jazyk	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
SPARQL	○	×	●	●	●	×	●	×	●	×	×
SeRQL	●	×	●	●	×	×	●	×	●	×	×
ARQ	●	●	●	●	●	●	●	●	●	●	×
Versa	●	○	○	×	●	●	●	●	×	●	×
XsRQL	○	×	●	●	●	●	●	×	●	×	×
TRIPLE	●	●	●	●	×	×	●	●	●	●	×
Xcerpt	●	●	●	●	●	●	●	●	●	●	×
Tequila	●	●	●	●	●	●	●	●	●	●	●

Legenda: ● umí vyjádřit, ○ umí vyjádřit pouze částečně, × neumí

Tabulka 5.1: Souhrn porovnání jazyků

Dotazy č. 5 a 6: Jazyky ARQ, Versa, Xcerpt, XsRQL a Tequila obsahují podporu pro agregační funkce. Jazyk SPARQL sice agregační funkce nemá, ale dotaz č. 5 je schopen vyřešit pomocí klauzulí `ORDER BY` a `LIMIT`. Dokument popisující jazyk XsRQL se sice vysloveně nezmiňuje o funkci `max`, potřebné pro vyjádření dotazu č. 5, ale návrh jazyka nebrání tuto funkci do jazyka přidat. Jazyk ARQ také zatím funkci `max` nemá, dotaz č. 5 je ale schopen vyřešit stejným způsobem jako jazyk SPARQL. Přidání agregačních funkcí do jazyka SeRQL je v plánu.

Dotaz č. 7: Tento dotaz lze pojmout různě. SPARQL, SeRQL, ARQ a XsRQL mohou vytvořit trojice, kde vlastnosti jedné z knih přiřadí druhé a naopak, tyto grafy ale opět nemohou použít v dotazech. Jazyk Tequila je schopen je vytvořit a použít. Jazyky Xcerpt a TRIPLE jsou schopny odvozováním popsat, že vlastnosti jedné knihy má i kniha druhá. Jazyk Versa je schopen vytvořit množinu s uzly obou knih a tu použít v přechodech, čímž docílí stejného efektu.

Dotaz č. 8: Tento dotaz jsou schopny vyjádřit jazyky TRIPLE, ARQ, Xcerpt a Tequila. Dále také jazyk SeRQL, ten ale nezvládne udělat uzávěr libovolné relace. Jazyk Versa nedovede vytvořit uzávěr relace ve formě grafu, ale je schopen po vlastnostech tranzitivně přejít.

Dotaz č. 10: Výběr prvků seznamu zvládnou jazyky SPARQL, SeRQL a XsRQL jen do určité hloubky. Zbylé jazyky dokáží vyjádřit dotaz na všechny prvky seznamu. Výběrem prvků ze seznamu se podrobně zabývala podkapitola 2.10.

Dotaz č. 11: Schopnosti provést vytvoření seznamu zvládne jen jazyk Tequila.

Dosažené výsledky jsou shrnuty v tabulce 5.1. Jazyk Tequila zvládl všechny ukázkové dotazy. Jako velmi schopné se dále jeví jazyky ARQ a Xcerpt. Všechny tyto jazyky obsahují agregační funkce[3]. Jejich další síla je dána použitím regulárních cest (jazyk ARQ), odvozování (jazyk Xcerpt) nebo pojmenovaných vzorů (jazyk Tequila). V čem však selžou všechny zde porovnávané jazyky kromě jazyka Tequila, to je schopnost vytvoření složitějšího grafu.

[3]V jazyce Tequila jsou implementovány pomocí pojmenovaného vzoru.

Kapitola 6

Pilotní implementace

Tato kapitola stručně popisuje pilotní implementaci překladače jazyka Tequila. Začíná popisem výchozího stavu zdrojového kódu úložiště Trisolda a popisuje úpravy, které bylo nutné provést, aby bylo možné dotazy jazyka Tequila v datovém úložišti Trisolda vyhodnocovat.

6.1 Trisolda a dotazy

Trisolda obsahuje podporu pro vyhodnocování dotazů, které odpovídají dotazům zapsaným v relačním kalkulu. Dotaz je reprezentován jako strom instancí potomků třídy `RDF_repository::Table`, z nichž každá reprezentuje tabulku. Listy stromu reprezentují tabulky, které se získají přímo z databáze, ostatní vrcholy stromu pak tabulky, které se získají nějakou operací relačního kalkulu. Trisolda obsahuje následující potomky třídy `Table`:

- `RDF_repository::BasicGraph` popisuje výběr dat z databáze na základě vzorů trojic.
- `RDF_repository::NaturalJoin` popisuje přirozené spojení tabulek.
- `RDF_repository::LeftNaturalJoin` popisuje levé přirozené spojení tabulek.
- `RDF_repository::Union` popisuje sloučení tabulek.
- `RDF_repository::Filter` popisuje filtrování obsahu tabulky.

Veškerá činnost s instancemi třídy `Table` je prováděna pomocí visitorů[14], kteří jsou potomky třídy `RDF_repository::TableVisitor`. Zdroj dat je reprezentován instancí potomka třídy `FlexibleEvaluator::Engine`. Trisolda obsahuje podporu pro tyto zdroje dat:

- `FlexibleEvaluator::BerkeleyEngine` reprezentuje data z databáze Berkeley.

- `FlexibleEvaluator::MemoryEngine` reprezentuje data uložená v paměti.
- `FlexibleEvaluator::NOPEngine` reprezentuje prázdný zdroj dat.
- `FlexibleEvaluator::OracleEngine` reprezentuje data z databáze Oracle.

Instance třídy `Engine` obsahují metodu `FindEvaluation`, která pro daný zdroj dat převádí dotaz popsaný pomocí instancí třídy `Table` na strom instancí potomků třídy `FlexibleEvaluator::EvaluationNode`, kteří zajišťují samotné vyhodnocení dotazu. Trisolda obsahuje tyto potomky třídy `EvaluationNode`:

- `FlexibleEvaluator::Impl::BerkeleyNode` vybírá data z databáze Berkeley na základě vzoru trojice. Tento uzel použije zdroj dat `BerkeleyEngine` k vyhodnocení tabulky `BasicGraph`.
- `FlexibleEvaluator::Impl::BookmarkNode` umožňuje vkládat do posloupnosti řešení záložku a následně se k ní vracet.
- `FlexibleEvaluator::Impl::FileNode` vybírá data ze souboru na základě vzoru trojic.
- `FlexibleEvaluator::Impl::FilterNode` umožňuje filtrovat řešení poduzlu na základě dané podmínky. Slouží k vyhodnocení tabulky `Filter`.
- `FlexibleEvaluator::Impl::MergeNode` provádí přirozené spojení a nebo levé přirozené spojení řešení svých dvou poduzlů. Slouží k vyhodnocení tabulek typu `NaturalJoin` a `LeftNaturalJoin`.
- `FlexibleEvaluator::Impl::ProjectionNode` provádí projekci řešení na dané proměnné.
- `FlexibleEvaluator::Impl::SortNode` třídí řešení svého poduzlu podle daných proměnných (sloupců).
- `FlexibleEvaluator::Impl::UnionNode` provádí spojení řešení svých poduzlů. Slouží k vyhodnocení tabulky `Union`.
- `FlexibleEvaluator::MemoryEngineImpl::MemoryNode` vybírá data z databáze v paměti na základě vzoru trojice. Tento uzel použije `MemoryEngine` k vyhodnocení tabulky `BasicGraph`.
- `FlexibleEvaluator::OracleEngineImpl::OracleNode` vybírá data z databáze Oracle na základě vzoru trojice. Tento uzel použije `OracleEngine` k vyhodnocení tabulky `BasicGraph`.
- `FlexibleEvaluator::OracleEngineImpl::SQLNode`

Každá instance `EvaluationNode` má dvě metody sloužící k jejímu vyhodnocování. Metodou `Next` se testuje, zda existuje další řešení, a metodou `Data` lze pak toto řešení získat.

Řešení je reprezentováno instancí třídy `FlexibleEvaluator::Tuple`, která odpovídá vektoru hodnot. Názvy jednotlivých položek lze získat z instance `EvaluationNode` metodou `Cols`.

6.2 Úpravy Trisoldy

Relační kalkul nabízený Trisoldou je pro vyhodnocování dotazů jazyka Tequila nedostačující (viz podkapitola 3.3), proto bylo třeba část Trisoldy přepracovat. Snahou při pilotní implementaci nicméně bylo co nejvíce využít stávající kód.

Byli přidáni další potomci třídy `Table`, tak aby bylo možné popsat dotazy jazyka Tequila. Přidáni byli následují potomci:

- `RDF_repository::TQLJoin` spojuje podvzory vzoru grafu.
- `RDF_repository::Match` popisuje vzor match.
- `RDF_repository::From` popisuje vzor from.
- `RDF_repository::Any` popisuje vzor any.
- `RDF_repository::Empty` popisuje prázdný vzor.
- `RDF_repository::Source` popisuje vzor source.
- `RDF_repository::NamedPattern` popisuje použití pojmenovaného vzoru.
- `RDF_repository::GetTriple` popisuje výběr trojice z databáze. K tomuto účelu nebylo možné použít třídu `BasicGraph`, neboť vzor trojice v jazyce Tequila může obsahovat také výrazy.
- `RDF_repository::AddTriple` popisuje vytvoření trojice.
- `RDF_repository::SubQuery` popisuje použití poddotazu.

Instance třídy `Tuple` byly rozšířeny o RDF graf tvořící vlastní řešení. Původní řešení, tj. vektor hodnot, nyní reprezentuje pouze vazbu proměnných na hodnoty. Graf je reprezentován instancí třídy `RDF_repository::Graph`. Aby bylo možné využít tento graf jako vstup pro další vzory, byl přidán zdroj dat `FlexibleEvaluator::GraphEngine` používající jako zdroj dat právě tento graf.

Aby bylo možné vyhodnocovat dotazy s rozšířenou sadou tabulek, bylo nutné přidat i další potomky třídy `EvaluationNode`:

- `FlexibleEvaluator::Impl::ConstantNode` představuje uzel s jediným řešením tvořeným daným grafem. Slouží k vyhodnocení tabulek `Empty` a `AddTriple`.
- `FlexibleEvaluator::Impl::ErrorNode` představuje uzel nevracející žádné řešení. Na tento uzel se vyhodnotí tabulky, při jejichž převodu došlo k nějaké chybě.
- `FlexibleEvaluator::Impl::FirstDataNode` vrací pouze první řešení svého poduzlu (má-li nějaké). Slouží k vyhodnocení tabulky `Any`.
- `FlexibleEvaluator::Impl::FromNode` slouží k vyhodnocení tabulky `From`. Postupně vyhodnocuje pravý podvzor, který má již převeden na strom instancí třídy `EvaluationNode`. Pro každé jeho řešení vytvoří `GraphEngine` a použije ho k vyřešení levého podvzoru, jehož řešení jsou i řešeními celého uzlu.
- `FlexibleEvaluator::Impl::GraphNode` vybírá data z grafu na základě vzoru trojice. Tento uzel použije `GraphEngine` k vyhodnocení tabulky `BasicGraph`.
- `FlexibleEvaluator::Impl::MergeGraphNode` sloučí řešení svého poduzlu do jednoho grafu. Používá se při vyhodnocování vnořeného dotazu.
- `FlexibleEvaluator::Impl::RenameNode` přejmenovává proměnné. Používá se při vyhodnocování pojmenovaných vzorů, kdy je třeba v závěru přejmenovat lokální proměnné pojmenovaného vzoru na výstupní proměnné.
- `FlexibleEvaluator::Impl::StripGraphNode` odstraní z řešení svého poduzlu grafy. Slouží k vyhodnocení tabulky `Match`.
- `FlexibleEvaluator::Impl::TQLMergeNode` tvoří základ vyhodnocování dotazů jazyka Tequila. Právě tento uzel implementuje hledání řešení s návratem. Postupně vyhodnocuje levý podvzor, který má již převeden na strom instancí třídy `EvaluationNode`. Pro každé jeho řešení provede dosazení hodnot proměnných do pravého podvzoru, který stále drží ve formě stromu instancí třídy `Table`. K tomu mu slouží `FlexibleEvaluator::InstancingTableVisitor`. Poté převede tento instanciovaný podvzor na instance `EvaluationNode` a postupně ho vyhodnocuje. Celkové řešení představuje spojení řešení levého a pravého podvzoru.

Dále bylo třeba doimplementovat podporu pro vyhodnocování výrazů, která nebyla v Trisoldě úplná.

6.3 Implementace překladače

Po provedení úprav Trisoldy je úkolem vlastního překladače už pouze vytvořit na základě textové reprezentace dotazu jeho reprezentaci pomocí instancí třídy `Table` a tuto vyhodnotit nad nějakým zdrojem dat.

K implementaci překladače byly použity nástroje `flex` (pro lexikální analýzu) a `bison` (pro syntaktickou analýzu).

6.4 Omezení pilotní implementace

Pilotní implementace obsahuje několik nedostatků, jako například:

- Z XML Schema datatypů jsou zatím podporovány pouze datatypy `xsd:string`, `xsd:boolean` a `xsd:integer`. Podporu pro další datatypy lze však snadno přidat.
- Vstupní data musí být v Berkeley databázi. Při vývoji překladače nebyla data v jiných zdrojích k dispozici.
- Vzor source umožňuje načíst data pouze ze souboru.

Kapitola 7

Závěr

Tato práce poskytla přehled hlavních myšlenek používaných v dotazovacích jazycích pro RDF a provedla jejich stručné zhodnocení. Pokusila se vybrat ty perspektivnější z nich a použít je při návrhu nového dotazovacího jazyka Tequila. Jako nové lze v tomto jazyce označit použití pojmenovaných vzorů, které umožnily nejen dosáhnout srovnatelných výsledků s ostatními jazyky co se týče výběru dat, ale zároveň je i předčít v schopnostech nová data vytvářet. Daní za to byla ovšem možnost zapsat dotazy, které není možné v konečném čase vyhodnotit. Rovněž pilotní implementace překladače tohoto jazyka pro datové úložiště Trisolda byla úspěšná.

Možnost další práce je hlavně v několika oblastech. První z nich je pokusit se navrhnout vhodnou sadu operátorů pro použití ve výrazech, který by umožnily podporovat širší škálu datatypů, nejen těch zabudovaných do jazyka. Navrhované operátory by měly umožnit například řetězcové operace, obecnou konverzi datatypů, práci v různých číselných soustavách atd.

Další práce by se mohly zabývat způsoby a možnostmi efektivního vyhodnocování dotazů jazyka Tequila tak, aby se například zabránilo zbytečnému explicitnímu vytváření velkých grafů. To se týká hlavně těch případů, kdy kvůli zabránění zacyklení je stejný vzor rekurzivně volán na stejná vstupní data, z nichž byla odstraněna jedna trojice.

Zajímavé by mohlo být rozšíření jazyka o podporu RDF datasetů. Použití této techniky by mohlo omezit výše zmíněné problémy, neboť hrany, které již byly zpracovány, by se umisťovaly do speciálního pojmenovaného grafu a nebylo by tak nutné v mnohých případech vytvářet rozsáhlé vstupní grafy.

Dodatek A

Agregační funkce

Tento dodatek popisuje možnosti jazyka Tequila při implementaci agregačních funkcí. Agregační funkce lze pojmout jako pojmenované vzory počítající agregační hodnoty ze svých vstupních grafů. Například pojmenovaný vzor `tql:count` může počítat počet trojic ve vstupním grafu, pojmenovaný vzor `tql:sum` součet objektů trojic ve vstupním grafu, pojmenovaný vzor `tql:max` maximální objekt z trojic ve vstupním grafu a pojmenovaný vzor `tql:avg` průměr objektů trojic ve vstupním grafu.

Možnou implementaci pojmenovaného vzoru `tql:max` ukazuje příklad A.1. Tento pojmenovaný vzor je tvořen vzorem union (řádek 25), jehož druhá větev uspěje jen nad prázdným grafem a vytvoří maximální hodnotu prázdného grafu jako prázdný literál. První větev díky vzoru any (řádek 6) uspěje jen nad neprázdným grafem a rekurzivně volá pojmenovaný vzor `tql:max` (řádek 9) nad vstupním grafem, z něhož byla odstraněna trojice získaná právě vzorem any (řádek 6). Výsledek pojmenovaného vzoru je svázán s proměnnou `?max` (řádek 8). Následný vnořený vzor union (řádek 19) vytvoří jako výsledek buď hodnotu proměnné `?obj`, pokud hodnota proměnné `?obj` je větší než hodnota proměnné `?max` a nebo hodnota proměnné `?max` odpovídá hodnotě prázdného grafu, a nebo hodnotu proměnné `?max` v opačném případě.

Implementaci pojmenovaných vzorů `tql:count` a `tql:sum` ukazují příklady A.2 a A.3, způsob jejich implementace je podobný příkladu A.1. Pojmenovaný vzor `tql:avg` je implementován v příkladu A.4 za použití předchozích dvou pojmenovaných vzorů. Jazyk Tequila obsahuje zatím pouze jednoduchou podporu výrazů, která neobsahuje možnost konverze datatypů. Z toho plyne omezení, že průměrem hodnot typu `typu xsd:integer` je opět hodnota typu `typu xsd:integer`.

Implementace agregačních funkcí je uložena v souboru `aggregate.tql`, odkud může být snadno načtena direktivou `import`.

```
prefix tql: <http://ulita.ms.mff.cuni.cz/tequila/term#>

tql:max()
{
    {
        match any ?subj ?pred ?obj.

        match ?max. from
        use tql:max() from get
        {
            ?X ?Y ?Z.
            filter !(?X = ?subj && ?Y = ?pred && ?Z = ?obj).
        }

        {
            filter ?obj > ?max || ?max = "".
            construct ?obj.
        }
        union
        {
            filter ?obj <= ?max && ?max != "".
            construct ?max.
        }
    }
    union
    {
        filter !get{?X ?Y ?Z.}.

        construct "".
    }
}
```

Příklad A.1: Implementace pojmenovaného vzoru `tql:max`

```
prefix tql: <http://ulita.ms.mff.cuni.cz/tequila/term#>

tql:count()
{
    {
        match any ?subj ?pred ?obj.

        match ?subcount. from
        use tql:count() from get
        {
            ?X ?Y ?Z.
            filter !(?X = ?subj && ?Y = ?pred && ?Z = ?obj).
        }

        construct (1 + ?subcount).
    }
    union
    {
        filter !get{?X ?Y ?Z.}.

        construct 0.
    }
}
```

Příklad A.2: Implementace pojmenovaného vzoru `tql:count`

```
prefix tql: <http://ulita.ms.mff.cuni.cz/tequila/term#>

tql:sum()
{
    {
        match any ?subj ?pred ?obj.

        match ?subsum. from
        use tql:sum() from get
        {
             ?X ?Y ?Z.
             filter !(?X = ?subj && ?Y = ?pred && ?Z = ?obj).
        }

        construct (?obj + ?subsum).
    }
    union
    {
        filter !get{?X ?Y ?Z.}.

        construct 0.
    }
}
```

Příklad A.3: Implementace pojmenovaného vzoru `tql:sum`

```
prefix tql: <http://ulita.ms.mff.cuni.cz/tequila/term#>
tql:avg()
{
    construct (get {use tql:sum()} / get {use tql:count()}).
}
```

Příklad A.4: Implementace pojmenovaného vzoru `tql:avg`

Dodatek B

Základní podpora pro RDF Schema

Možnou základní podporu pro RDF Schema lze kromě dotazu, který do vstupních dat doplní trojice plynoucí z rdfs: odvození, jehož výsledek se použije jako vstup pro samotný dotaz, řešit také implementací různých pojmenovaných vzorů obstarávajících základní rdfs: semantiku.

Příkladem může být pojmenovaný vzor `ex:subClassesOf`, který dostane jako parametr `?class` třídu a vrací tuto třídu a všechny její podtřídy. Tento pojmenovaný vzor lze použít například k otestování, zda nějaká instance je instancí dané třídy (viz příklad 4.3).

Implementaci pojmenovaného vzoru `ex:subClassesOf` ukazuje příklad B.1. Ten prostřednictvím první větve vzoru union (řádek 11) vrací svůj parametr (řádek 9). Pomocí druhé větve vzoru union projde přímé potomky třídy `?class` (řádek 13) a pro každého z nich zavolá rekurzivně pojmenovaný vzor `ex:subClassesOf`. Aby se předešlo zacyklení[1], je pro rekurzivní volání použit vstupní graf, z nějž byly odstraněny trojice obsahující třídu `?class` jako subjekt. Použití vnořeného dotazu (řádek 6) zajišťuje, že vzor má vždy právě jedno řešení obsahující vše v jednom grafu.

Implementace pojmenovaného vzoru je uložena v souboru `schema.tql`, odkud může být snadno načtena direktivou `import`.

[1] Dědičná hierarchie v RDF Schema může tvořit cyklus.

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix ex:   <http://www.example.org/term#>

ex:subClassesOf(?class)
{
    get
    {
        {
            construct ?class.
        }
        union
        {
            match ?subClass rdfs:subClassOf ?class.

            use ex:subClassesOf(?subClass) from get
            {
                ?subj rdfs:subClassOf ?obj.
                filter ?obj != ?class.
            }
        }
    }
}
```

Příklad B.1: Implementace pojmenovaného vzoru `ex:subClassesOf`

Dodatek C

Gramatika

Dodatek obsahuje gramatiku jazyka Tequila zapsanou v EBNF notaci [12].

```
Start              :=  Prefix* Import* Let* MainQuery
Prefix             :=  'prefix' PN_PREFIX URIREF
Import             :=  'import' STRING
Let                :=  Uri '(' (Variable (',' Variable)*)? ')' GrafPattern
MainQuery          :=  Query ('where' GrafPattern )? ( 'from' GrafPattern )?
Query              :=  'get' GrafPattern
GrafPattern        :=  '{' UnionPattern* '}'
UnionPattern       :=  FromPattern ( 'union' FromPattern )*
FromPattern        :=  (WherePattern 'from' )* WherePattern
WherePattern       :=  ( UnaryPattern 'where' )* UnaryPattern
UnaryPattern       :=  Triple
                       | Uniple
                       | 'filter' Expression '.'
                       | 'optional' UnaryPattern
                       | 'use' (Uri | Variable) '(' (Expression (',' Expression)*)? ')'
                       | 'construct' ('{' UniOrTriple* '}' | UniOrTriple)
                       | 'any' UnaryPattern
                       | GrafPattern
                       | Query
Triple             :=  PatternExp PatternExp PatternExp '.'
Uniple             :=  PatternExp '.'
UniOrTriple        :=  Uniple | Triple
Expression         :=  ConditionalOrExp
ConditionalOrExp   :=  ConditionalOrExp ('||' ConditionalAndExp)*
ConditionalAndExp  :=  ConditionalAndExp ('&&' RelationalExp)*
RelationalExp      :=  AdditiveExp (('=' | '!=' | '<' | '>' | '<=' | '>=' | 'like') AdditiveExp)?
AdditiveExp        :=  MultiplicativeExp (('+' | '-') MultiplicativeExp)*
MultiplicativeExp  :=  UnaryExp (('*' | '/' ) UnaryExp)*
UnaryExp           :=  ('!' | '+' | '-')? PrimaryExp
BasePrimaryExp     :=  Variable | Literal | Uri | BlankNode | '(' Expression ')'
PatternExp         :=  BasePrimaryExp | MinusLiteral
PrimaryExp         :=  BasePrimaryExp | MainQuery
Literal            :=  RDFLiteral | NumericLiteral BooleanLiteral
RDFLiteral         :=  STRING ( '^^' Uri | LANGTAG )?
NumericLiteral     :=  INTEGER | DECIMAL | DOUBLE
MinusLiteral       :=  '-' NumericLiteral
BooleanLiteral     :=  'true' | 'false'
Uri                :=  URIREF | PN_PREFIX ( PN_LOCAL ) ?
BlankNode          :=  '_:' PN_LOCAL
Variable           :=  IDENTIFIER
```

```
STRING          :=  STRING1 | STRING2 | STRING_LONG1 | STRING_LONG2
STRING1         :=  "'" ( ([^'\#xA#xD]) | ECHAR )* "'"
STRING2         :=  '"' ( ([^'\#xA#xD]) | ECHAR )* '"'
STRING_LONG1    :=  "'''" ( ( "'" | "''" )? ( [^'\] | ECHAR ) )* "'''"
STRING_LONG2    :=  '"""' ( ( '"' | '""' )? ( [^"\] | ECHAR ) )* '"""'
ECHAR           :=  '\' ('t' | 'b' | 'n' | 'r' | 'f' | '\' | '"' | "'")
IDENTIFIER      :=  ('$' | '?') ALPHA (ALPHA | DIGIT)*
LANGTAG         :=  '@' ALPHA+ ('-' (ALPHA | DIGIT)+)*
INTEGER         :=  DIGIT+
DECIMAL         :=  DIGIT+ '.' DIGIT* | '.' DIGIT+
DOUBLE          :=  DIGIT+ '.' DIGIT* EXPONENT | '.' (DIGIT)+ EXPONENT | (DIGIT)+ EXPONENT
EXPONENT        :=  ('e' | 'E') ('+' | '-')? DIGIT+
PN_PREFIX       :=  ALPHA ((ALPHA | DIGIT | '_' | '-' | '.')* (ALPHA | DIGIT | '_' | '-'))? :
PN_LOCAL        :=  (ALPHA | '_') ((ALPHA | DIGIT | '_' | '-' | '.')* (ALPHA | DIGIT | '_' | '-'))?
URIREF          :=  '<' URI '>'
URI             :=  SCHEME ':' HIER_PART ('?' QUERY)? ('#' FRAGMENT)?
HIER_PART       :=  ('//' AUTHORITY PATH_ABEMPTY) | PATH_ABSOLUTE | PATH_ROOTLESS
SCHEME          :=  ALPHA (ALPHA | DIGIT | '+' | '-' | '.')*
AUTHORITY       :=  (USERINFO '@')? HOST (':' PORT)?
USERINFO        :=  (UNRESERVED | PCT_ENCODED | SUB_DELIMS | ':')?
HOST            :=  IP_LITERAL | IPV4ADDRESS | REG_NAME
PORT            :=  DIGIT*
IP_LITERAL      :=  '[' (IPV6ADDRESS | IPVFUTURE) ']'
IPVFUTURE       :=  'v' HEXDIG+ '.' (UNRESERVED | SUB_DELIMS | ':')+
IPV6ADDRESS     :=  H16 ':' H16 ':' H16 ':' H16 ':' H16 ':' H16 ':' LS32
                    | '::' H16 ':' H16 ':' H16 ':' H16 ':' H16 ':' LS32
                    | H16? '::' H16 ':' H16 ':' H16 ':' H16 ':' LS32
                    | ((H16 ':')? H16)? '::'H16 ':'H16 ':'H16 ':' LS32
                    | ((H16 ':')? (H16 ':')? H16)? '::'H16 ':'H16 ':' LS32
                    | ((H16 ':')? (H16 ':')? (H16 ':')? H16)? '::'H16 ':' LS32
                    | ((H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? H16)? '::' LS32
                    | ((H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? H16)? '::' H16
                    | ((H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? (H16 ':')? H16)? '::'
H16             :=  HEXDIG HEXDIG? HEXDIG? HEXDIG?
LS32            :=  (H16 ':' H16) | IPV4ADDRESS
IPV4ADDRESS     :=  DEC_OCTET '.' DEC_OCTET '.' DEC_OCTET '.' DEC_OCTET
DEC_OCTET       :=  DIGIT | ([1-9] DIGIT) | ('1' DIGIT DIGIT) | ('2' [0-4] DIGIT) | ('25' [0-5])
REG_NAME        :=  (UNRESERVED | PCT_ENCODED | SUB_DELIMS)*
PATH_ABEMPTY    :=  ('/' SEGMENT)*
PATH_ABSOLUTE   :=  '/' (SEGMENT_NZ ('/' SEGMENT)*)?
PATH_ROOTLESS   :=  SEGMENT_NZ ('/' SEGMENT)*
SEGMENT         :=  PCHAR*
SEGMENT_NZ      :=  PCHAR+
PCHAR           :=  UNRESERVED | PCT_ENCODED | SUB_DELIMS | ':' | '@'
QUERY           :=  (PCHAR | '/' | '?')*
FRAGMENT        :=  (PCHAR | '/' | '?')*
PCT_ENCODED     :=  '%' HEXDIG HEXDIG
UNRESERVED      :=  ALPHA | DIGIT | '-' | '.' | '_' | ' '
RESERVED        :=  GEN_DELIMS | SUB_DELIMS
GEN_DELIMS      :=  ':' | '/' | '?' | '#' | '[' | ']' | '@'
SUB_DELIMS      :=  '!' | '$' | '&' | "'" | '(' | ')' | '*' | '+' | ',' | ';' | '='
ALPHA           :=  [A-Za-z]
DIGIT           :=  [0-9]
HEXDIG          :=  [0-9A-F]
```

Dodatek D

Obsah CD

Na přiloženém CD se nachází tyto soubory:

`thesis.pdf` – elektronická verze této diplomové práce.

`tequila-0.9.tar.gz` – zdrojové kódy pilotní implementace.

Archív `tequila-0.9.tar.gz` obsahuje tyto adresáře a soubory:

`import` – import soubory.

`queries` – ukázkové dotazy.

`semweb` – upravené zdrojové kódy Trisoldy.

`src` – zdrojové kódy překladače jazyka Tequila.

`ulibpp` – zdrojové kódy knihovny ULIB++.

`README` – popis instalace a použití.

Literatura

[1] ARQ - Documentation and Resources.
URL http://jena.sourceforge.net/ARQ/documentation.html

[2] ARQ - Property Paths.
URL http://jena.sourceforge.net/ARQ/property_paths.html

[3] Haskell 98 Language and Libraries, The Revised Report.
URL http://haskell.org/definition/haskell98-report.pdf

[4] Trisolda.
URL http://ulita.ms.mff.cuni.cz/Trisolda/

[5] User Guide for Sesame,Chapter 6. The SeRQL query language (revision 1.2).
URL http://www.openrdf.org/doc/sesame/users/ch06.html

[6] RQL v2.1 User Manual. Červenec 2003.
URL http://139.91.183.30:9090/RDF/RQL/Manual.html

[7] RDF Primer, W3C Recommendation. Únor 2004.
URL http://www.w3.org/TR/rdf-primer/

[8] RDF Semantics, W3C Recommendation. Únor 2004.
URL http://www.w3.org/TR/2004/REC-rdf-mt-20040210/

[9] RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation. Únor 2004.
URL http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[10] RDF/XML Syntax Specification (Revised), W3C Recommendation. Únor 2004.
URL http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/

[11] Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation. Únor 2004.
URL http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[12] Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation. Září 2006.
URL http://www.w3.org/TR/2004/REC-xml11-20040204/

[13] SPARQL Query Language for RDF, W3C Recommendation. Leden 2008.
URL http://www.w3.org/TR/rdf-sparql-query/

[14] Visitor pattern. Červenec 2008.
URL http://en.wikipedia.org/wiki/Visitor_pattern

[15] Berners-Lee, T.; Fielding, R.; Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), Leden 2005.
URL http://www.ietf.org/rfc/rfc3986.txt

[16] Berners-Lee, T.; Hendler, J.; Lassila, O.: The Semantic Web—A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, Květen 2001.

[17] Bolzer, O.: *Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt*. Diplomová práce, Institute of Computer Science, LMU, Munich, 2005.
URL http://www.pms.ifi.lmu.de/publikationen/#DA_Oliver.Bolzer

[18] Furche, T.; Bry, F.; Schaffert, S.: Xcerpt 2.0 Specification of the (Core) Language Syntax. Duben 2007.
URL http://rewerse.net/deliverables/m36/i4-d12.pdf

[19] Furche, T.; Linse, B.; Bry, F.; aj.: RDF Querying: Language Constructs and Evaluation Methods Compared. 2006.
URL http://rewerse.net/publications/download/REWERSE-RP-2006-071.pdf

[20] Haase, P.; Broekstra, J.; Eberhart, A.; aj.: A Comparison of RDF Query Languages.
URL http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf

[21] Katz, H.: XsRQL: an XQuery-style Query Language for RDF. Červen 2004.
URL http://www.fatdog.com/xsrql.html

[22] Olson, M.; Ogbuji, U.: Versa.
URL http://acs.lbl.gov/~ksb/4Suite-1.0a1-docs/html/Versa.html

[23] Schaffert, S.: Xcerpt Tutorial. Březen 2003.
URL http://www.xcerpt.org/documentation/tutorial

[24] Sintek, M.; Decker, S.: TRIPLE—A Query, Inference, and Transformation Language for the Semantic Web.
URL http://triple.semanticweb.org/doc/iswc2002/TripleReport.pdf